SOURCE LEVEL DEBUGGING OF DYNAMICALLY TRANSLATED PROGRAMS

by

Naveen Kumar

B.Tech., Institute of Technology, India, 2000

M.S., University of Pittsburgh, 2002

Submitted to the Graduate Faculty of

University of Pittsburgh in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Naveen Kumar

It was defended on

March 24, 2008

and approved by

Dr. Bruce R. Childers, Associate Professor, University of Pittsburgh

Dr. Mary Lou Soffa, Professor, University of Virginia

Dr. Sangyeun Cho, Assistant Professor, University of Pittsburgh

Dr. Mike Smith, Professor, Harvard University

Thesis Advisor: Dr. Bruce R. Childers, Associate Professor, University of Pittsburgh

Thesis Advisor: Dr. Mary Lou Soffa, Professor, University of Virginia

Acknowledgements

I am indebted to my advisors, Bruce Childers and Mary Lou Soffa, for their support, encouragement, friendship and guidance throughout my graduate studies. Without them, this dissertation would not have been possible. I am also grateful to Sangyeun Cho and Mike Smith, who took the time to be on my committee and provided suggestions concerning this research.

I cannot thank my parents, Subhadra Devi and Sushil Jha, enough for their overwhelming and unconditional support and love. It is their faith in my ability to achieve my goals that provided me the motivation to pursue and complete my doctorate. I would like to thank my siblings Bharti, Arun and Pravin for always being there for me and believing in me more than I do. Finally, I would like to thank my wife, Ritu, for her affection, motivation, and most of all, putting up with me during the past year.

To my parents

Subhadra Devi and Sushil Jha

SOURCE LEVEL DEBUGGING OF DYNAMICALLY TRANSLATED PROGRAMS

Naveen Kumar, Ph.D.

University of Pittsburgh, 2008

The capability to debug a program at the source level is useful and often indispensible. Debuggers use sophisticated techniques to provide a source view of a program, even though what is executing on the hardware is often machine code. The techniques used by debuggers continuously evolve with changes in programming language strategies and execution environments. Recently, a new execution mechanism has emerged, called software dynamic translation, that inserts a layer of software between a program and a host machine. The software layer has complete control of an executing application, which is then modified for a specific purpose. A spectrum of diverse uses of software dynamic translation have emerged, including dynamic optimization, dynamic instrumentation, software security checking, host machine virualization and simulations, among others. With the growing number of uses of software dynamic translation in academia and industry, a lack of adequate source level debugging techniques is currently a stumbling block.

This thesis is the first research that targets source level debugging of dynamically translated programs. The techniques developed in this research do not require or induce changes in the program when it is being debugged. In other words, the program being debugged is the same as that outside of the debug session. The debugging techniques are not catered to a specific SDT system or implementation; they can be applied to a broad range of SDT systems.

With software dynamic translation, a program's code is modified at runtime, including when it is being debugged. Therefore, the debugger's view of the program must be kept consistent with the changing program for debugging accuracy. This thesis develops a framework, called Tdb, that generates and maintains consistent information about the constantly changing program and uses the information for debugging.

Tdb's organization and its techniques offer several advantages. First, Tdb allows debug users to choose their favorite debugger. Users do not have to learn new commands or be aware that the program being debugged is being dynamically translated. Second, Tdb enables different dynamic translators to be used with the same debugger, thereby providing portability to its techniques across dynamic translators.

This thesis describes the Tdb framework and experimentally evaluates its techniques. Tdb's techniques are implemented for three dynamic translators on two different hardware platforms and experiments are conducted. The experimental results demonstrate that source level debugging of dynamically translated programs is possible. Further, the experiments show that Tdb's techniques are practical as well as efficient.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

Programmatic errors, or bugs, are unavoidable during the development of all but the simplest of programs. The importance of identifying and eliminating bugs cannot be stressed enough. A debugger is perhaps the most important tool in a programmer's repertoire for locating bugs. A debugger allows programs to be executed in a controlled manner so that a programmer can inspect values computed and paths taken during execution. With the aid of a debugger, a programmer can track bugs in the program (i.e., debug) by pinpointing the location where an incorrect program path is taken or incorrect value is computed.

Debuggers often provide a source view of the program being debugged. This means, even though the program executable may be represented as binary code or intermediate code, such as bytecode [55], a debugger still allows a programmer to control or inspect the program's execution from the point of view of source code constructs. For example, a programmer may specify a source line number at which the debugger should pause the execution. The debugger finds the corresponding location in the binary code or bytecode and pauses execution at that location. In this way, a debugger hides machine-specific details from programmers.

Relating source code with binary code becomes complicated when the binary code does not directly correspond with the source code. Consider compiler optimizations for example. Optimizations make code transformations, such as code movement, code deletion, code addition and register allocation. With code transformations, values may be computed earlier or later in the executing program than expected by the programmer and the order in which statements execute can change with respect to the original source code. In addition to hiding machine-specific details, the debugger also has to now hide the effect of optimizations from debug users. In particular, debuggers have to solve two problems: (1) locating statements in transformed code, called the *code location problem*, and (2) extracting variable values which are not available at expected locations due to code transformations, called the *data value problem*. To solve these problems, techniques have been developed that use static as well as dynamic analyses to relate unoptimized code with optimized code and extract "expected" variable values for inspection [8,22,34,41,1,100,102].

A debugger's task becomes yet more complicated with concurrent programs. Identifying race conditions can be extremely difficult. Debugging typically involves pausing execution which can change timing dependencies. A debugger has to preserve these dependencies while allowing programmers to pause and inspect the program. Debugging techniques such as timestamping and checkpointing have been developed to address the concurrency challenges and have proved invaluable [17,73,103].

Optimizations and concurrency are only two examples where debugging technology had to evolve to address new challenges. In fact, with advances in code generation strategies and programming language paradigms, debuggers have always had to address new challenges in relating a source program with the executing program. Vector programming, object-oriented programming and just-in-time compilation are some of the other paradigm changes in execution strategies and programming environments where new debugging techniques were developed [39,53,67,69,43,70]. Recently, a new execution environment, software dynamic translation (SDT), has become increasingly popular. SDT provides exciting new capabilities for system programmers that were previously difficult or impossible to achieve. Currently, adequate source level debugging techniques do not exist for SDT systems. This thesis provides significant advancement of SDT technology and the debuggers by developing source level debugging techniques in this new execution domain. This thesis is the first work that provides debugging techniques generally applicable to any SDT system and that does not inhibit or change SDT system's behavior when a program is being debugged.

1.1 Software Dynamic Translation

A SDT system is a layer of software that mediates program execution by intercepting a program before it is run on the underlying hardware. The placement of the SDT system with respect to an application program and the underlying host machine is shown in Figure 1. SDT allows for monitoring the run-time behavior of an executing program and modifying its execution in a controlled manner.

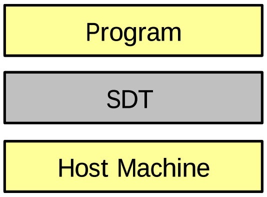


Figure 1: A software dynamic translator

The ability to monitor any executing program and modify the program's code and data gives system developers unprecedented power and flexibility. For example, SDT technology can be used to translate an instruction stream to a different instruction set architecture, thereby allowing a program to be run on a different processor than originally intended. This technique, called binary translation, is used by Rosetta for running applications compiled for the PowerPC family of processors on Intel processors [2]. Similarly, SDT technology is used by virtual machine monitors to intercept privileged instructions and replace them by their emulated counterpart [28,77]. SDT systems have become commonplace in research circles [3,5,9,11,14,18,20,47,64,80,85,95,96,107] and are gaining popularity in commercial circles [2,6,10,15,25,28,77,97]. SDT technology has been used in diverse domains, including security checking [20,47,80], dynamic code optimization [3,5,9,14,64,97], binary translation of one instruction set to another [6,15,18,25,28,85,95,96], host machine virtualization [11,45,55,76,98,101], execution in a memory constrained system [83,106,107], and computer architecture simulation [19,76,82,101].

Despite the increasing popularity of SDT technology, adequate source level debugging techniques do not exist for dynamically translated applications. Current SDT systems either do not provide any debug support or generate different code during a debug session than otherwise. SDT systems providing no debug support include commercial systems such as Rosetta [2] as well as research prototypes such as Dynamo [5]. Application developers must develop their applications and debug them without dynamic translation and then deploy them with the SDT system. At the other end of spectrum are SDT systems that generate different code when a breakpoint is inserted or a debugger is attached to a program. While such debug support is vastly more desirable than no debug support, it is less than ideal because the same code may be translated differently in different invocations of the SDT system. For example, different code transformations may be applied to a code block during retranslation when the program is debugged. Code transformations can expose or hide latent bugs in programs [34]. Therefore, it is inadequate to debug and test different code than what is actually deployed. To stress the importance of debugging in the deployment environment, Hennessy observed in a seminal paper on debugging optimized code: “The ability to debug optimized code symbolically and reliably is an important asset that should not be relinquished” [34]. Today, this observation is equally relevant in the context of dynamic translation. The lack of source level debugging techniques for SDT systems are due to several challenges, as discussed next.

1.2 Challenges to Debugging

Traditionally, compilers generate debug information that relates source code of a program with its executable. The debug information is stored with the executable program on disk. Debuggers use this information in a debug session to relate the executing program with the source code. With SDT systems, this traditional approach to debugging programs is no longer applicable because the executable program is modified at runtime, thereby rendering the static debug information inconsistent with the executing code. Specifically, there are three challenges that must be addressed in order to debug dynamically translated programs at the source level.

1.2.1 Code Transformation

The first challenge is solving the code location and data value problems that arise due to code transformations performed by SDT systems. SDT systems transform application code throughout a program's execution. To avoid inconsistency between the debug information and executing code, the debug information must be generated and modified simultaneously with the code.

Code transformations include insertion of new code, removal or modification of existing code, duplication of statement instances and a change in the order in which statements are executed. Additionally, SDT systems often flush away existing code segments and regenerate them. Code that has been transformed can be modified again and relocated to a different code segment. Code transformations may also lead to a change in location of a data value, via register allocation, and a change in the live range of a data value, via statement re-ordering.

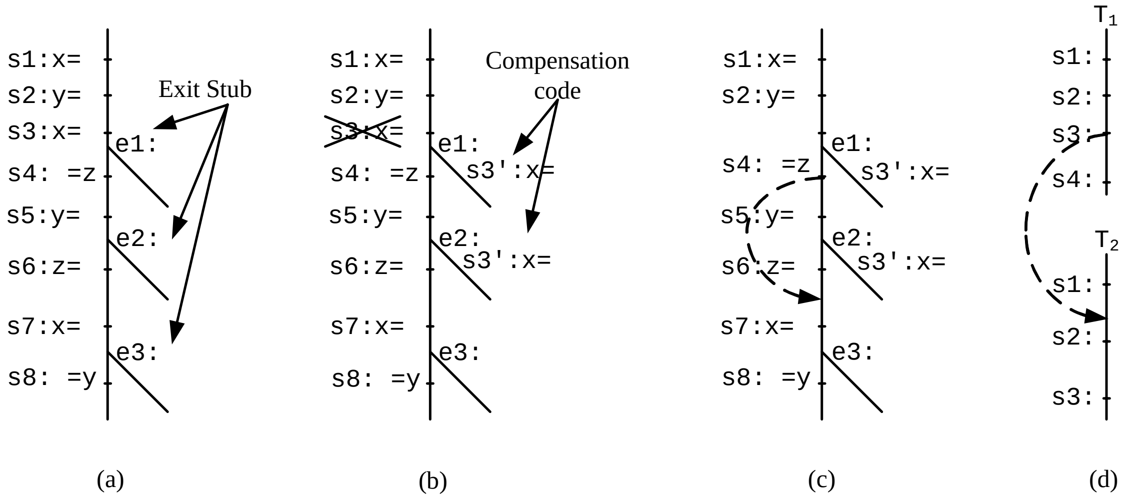


Figure 2: Challenges to debugging optimized instruction traces

Figure 2 uses an example debug scenario in the presence of a SDT system to illustrate the challenges posed by code transformation. In the example, a SDT system transforms a straightline code sequence over the course of its execution. Figure 2(a) depicts an *instruction trace*, which is a straightline single-entry, multiple-exit code sequence (statements `s1` through `s8`). Control transfers out of the trace are via *exit stubs* (`e1`, `e2` and `e3`). Figure 2(b) through Figure 2(d) show the result of applying code transformations in separate phases. In Figure 2(b), statement s3 is eliminated from its original location and inserted into two exit stubs and marked as “compensation code”. Suppose a breakpoint is inserted at s3 and s4 and the value of x is queried. If execution reaches s4, the associated breakpoint will pause execution. This behavior is not expected to the programmer who expects the program to pause at s3 and subsequently at s4, when execution is continued. In addition, the value of x reported at s4 will not be the expected value.

The problem illustrated in Figure 2(b), namely code movement, can be solved by techniques previously developed for debugging (statically) optimized code [8,22,34,41,1,100,102]. There are other code transformations frequently applied by SDT systems (e.g., code flush) that have not been addressed by prior work. But even if we assume that debug techniques are available to handle each code transformation performed by a SDT system, the very nature of SDT systems render previous solutions insufficient, as illustrated in Figure 2(c). Suppose debug techniques are available such that if a breakpoint is inserted at s3 and s4 in Figure 2(b) and value of x is queried, the debugger pauses execution twice and extracts the expected value of x and presents it to the programmer. During further execution, assume that statement s4 is later moved during a new transformation phase as shown in Figure 2(c). The debugging techniques for optimized code will relate code in Figure 2(c) with that in Figure 2(b). If execution is paused at s4, then the debugger will assume that x’s value is available (computed by s1) because the debug information was not generated relative to the original code in Figure 2(a). In fact, the original code is deleted after the first transformation phase. The challenge posed by multiple levels of transformation is to relate transformed code with original code that is no longer available.

Figure 2(d) depicts another problem with code transformation in SDT systems where two traces T1 and T2 are combined and a statement is moved from T1 to T2. Code transformations are typically applied to a group of instructions at a time. For example, traditionally optimizations are applied at a method granularity. With code granularities changing throughout the execution, each instruction must be tracked independently even though modifications are applied to groups of instructions.

1.2.2 Online Communication

The second challenge to debugging dynamically translated programs is communicating debug information to a debugger as code transformations are applied. In addition, since the execution of SDT system is inter-

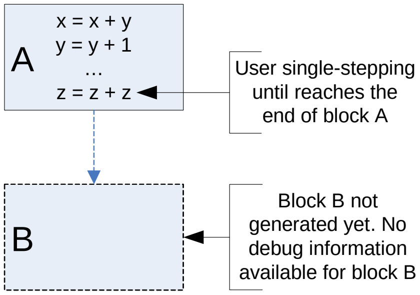

A
x = x + y
y = y + 1
…
z = z + z
User single-stepping until reaches the end of block A
B
Block B not generated yet. No debug information available for block B

Figure 3: Incremental generation of debug information. Block B has not been generated yet, so the debug information does not exist either.

leaved with that of the translated application, online information about when the application starts and stops executing should be made available to the debugger. With this information, the debugger can ensure that execution is paused only when the program is executing — i.e., not when the SDT system is executing.

Consider Figure 3 in which a debug user is single-stepping through basic block `A` in the program. The code for block `B` has not been generated. When execution reaches the last statement in block `A` and the next block to be executed is `B`, then the SDT system translates code for block `B`. As code is generated for `B`, corresponding debug information must be generated and immediately communicated to the debugger. If the translated code for `B` is later modified, the debug information needs to be updated. In general, debug information must be computed during translation as new code is generated, and updated when existing translated code is modified or deleted.

1.2.3 Efficiency of Debug Information Generation

The last challenge to debugging dynamically translated programs is providing the debug support with low overhead. Traditionally, the overhead of computing debug information was never a concern because the debug information was generated offline. With SDT systems, the overhead of computing debug information increases the runtime of the program. Ideally, debug information should always be generated so that a programmer has the flexibility to "attach" a debugger to a program that was originally started outside of a debug session. Another benefit of always generating debug information is that a programmer can perform post-mortem debugging by analyzing core dumps containing debug information. The overheads of computing debug information, therefore, must be minimal.

1.3 Research Overview

This dissertation research explores the following open problems associated with source level debugging in a SDT environment:

- Code location and data value problems in the context of SDT systems.
- Online communication of debug information to the debugger.
- Efficient generation and maintenance of runtime debug information.

This research develops solutions to the above problems and presents a new framework, called Tdb, which allows a dynamically translated program to be debugged at the source level. The techniques used by Tdb do not restrict code generation or transformation. At the same time, the code debugged in this framework is the same code that is deployed. The techniques developed in this research fulfill several requirements, discussed below, so that they can be useful to a wide audience.

- Portability: The techniques should be portable across different SDT systems and hosts.
- Usability: The techniques should not impose a steep learning curve to debug users. That is, the debugger should be friendly to use.
- Efficiency: The performance and memory overheads of the debugger should be comparable to that of traditional debuggers of statically generated code..

To solve the debugging problems in SDT systems while fulfilling the above requirements, Tdb provides the following solutions. Tdb proposes a new SDT component that tracks program transformations. A representation to describe program transformations is developed. Tdb also describes how the code location and data value problems can be solved by generating debug information, based upon the program transformations, for use by a debugger.

Online communication between a SDT system and a debugger is facilitated by means of well-defined Tdb interfaces. Operating system support, similar to those used by conventional debuggers, is used to perform the actual communication between the SDT system and the debugger.

Finally, for efficiency reasons, Tdb maintains fine-grained debug information that is easy to generate and modify. Also, the online communication mechanism used in Tdb does away with traditional file-based communication that would be very slow for continuous update and use of debug information. Further, SDT systems spend little overhead in code transformations because the time spent in transformation adds to the overall runtime of the program. Tdb uses algorithms that are fine-tuned to SDT systems so that tracking program transformations is extremely lightweight.

The techniques developed in this dissertation research have been primarily implemented in the Strata SDT infrastructure on SPARC platform [79]. Some of the techniques developed here have also been implemented in the Pin SDT system on x86 platform [59]. The techniques developed in this research are comprehensive and as such applicable to different SDT systems such as dynamic optimizers, security checkers and simulators, among others. To illustrate the usefulness of Tdb in diverse SDT systems, this thesis presents two source level debuggers based on Tdb: (1) a debugger for dynamically optimized programs, and (2) a debugger for dynamically instrumented programs.

1.4 Scope of the Thesis

This thesis assumes that a SDT system preserves the original semantics of the programs that it translates. It does not target SDT systems that modify the original semantics of a program. For example, a SDT system can be used as a software updater [35] that dynamically replaces program modules with their new versions.

This research does not target such SDT systems. Note that the above limitation does not rule out most SDT systems in existence today [3,5,9,11,14,18,20,47,64,80,85,95,96,107].

This thesis targets sequential programs that are represented in a binary format. This thesis does not solve debugging problems associated with programs in special domains, including concurrent programs, just-in-time compiled programs, distributed programs and real time programs.

Finally, this thesis focuses on source level debugging with of breakpoints. The debugging features are limited to insertion/removal of breakpoints and inspection of data values. A number of other debugging features can be built using these basic features, including single-stepping through code, tracing the stack frame, and watchpoints, among others. In fact, most of the source level commands of a popular debugger, Gdb [88], can be realized using the features targeted in this thesis.

1.5 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the background necessary to understand this work and the previous work that is related to this research and work that is used as a foundation in this dissertation. Chapter 3 provides an overview of the Tdb framework. Chapter 4 describes a representation for specifying program transformations in the Tdb framework. Chapter 5 discusses how information about program transformations can be used to facilitate source level debugging. This chapter also describes the communication mechanism between the SDT system and the debugger. Chapter 6 gives detailed descriptions of how the different components of Tdb can be realized in practice. This chapter discusses the choices that can be made during implementation. Chapter 8 and Chapter 7 provide the debuggers Tdb-1 and Tdb-3 to showcase how source level debugging is feasible using techniques developed in this research. Experimental evaluation of the two debuggers illustrate the efficiency of the debugging techniques. Conclusions and directions for future research are discussed in Chapter 9.

Chapter 2. Background and Related Work

Source level debuggers have existed for decades. However, the technology used by debuggers has evolved along with changes in programming paradigms. The techniques developed in this thesis build upon existing debugging research, which are discussed in this chapter. This chapter first describes, in Section 2.1, the background for understanding the execution environment presented by SDT systems. It also describes related work with SDT systems. In Section 2.2, the chapter gives background for source level debugging and the prior work which relates to this research.

2.1 Software Dynamic Translation: The Execution Environment

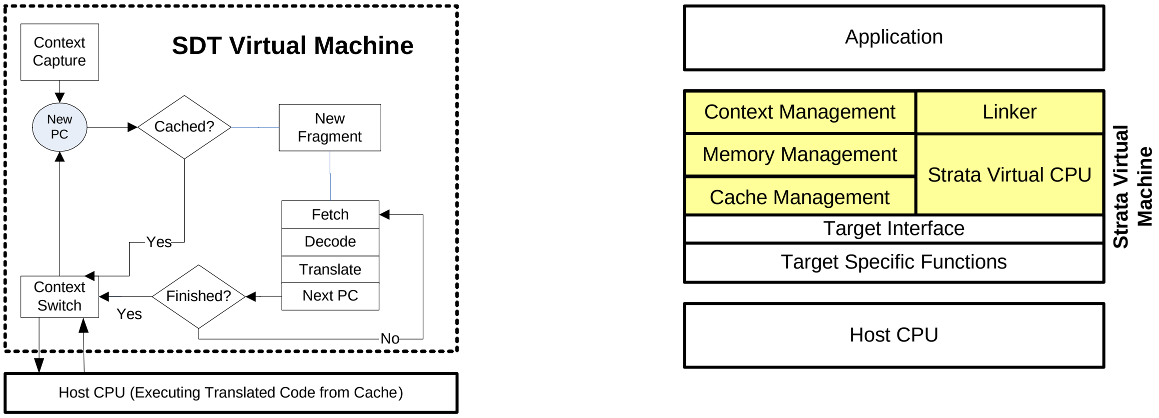
Software dynamic translation involves a virtual machine within which an application program executes. The virtual machine intercepts the program's instruction stream and potentially transforms it before the instructions are executed on real hardware. The ability to modify a program's instruction stream is an unprecedented flexibility that software dynamic translation offers to system designers. This flexibility allows system designers to accomplish a variety of objectives not easily achieved by other means. For instance, SDT systems such as Rosetta [2], Transmeta's CMS [25], Daisy [28] and FX!32 [15], can be used to overcome cost barriers to new platform acceptance via binary translation. VMware's virtual machine monitor uses SDT to replace privileged instructions with their emulated counterparts to achieve host machine virtualization [77]. Shade uses dynamic translation to implement high-performance instruction set simulators [19]. Embra uses it to implement a high-performance operating system emulator [101]. Dynamo and Mojo [5,14] use software dynamic translation to improve the performance of native binaries, and Daisy uses dynamic translation to evaluate the performance of novel VLIW architectures and accompanying optimization techniques [28]. And recently software dynamic translation has been used to ensure safe execution of untrusted binaries [47].

This section describes how SDT systems work and gives insight into how new SDT systems can be realized. This section first describes, in Section 2.1.1, the structure and functionality of a *basic SDT system* that provides a virtual machine for executing a program but does not modify the program's instruction stream. The next section, Section 2.1.2, describes several overhead reduction techniques that are typically applied by SDT systems to reduce the runtime overheads associated with software virtualization. Finally, Section 2.1.3 uses four diverse SDT systems to illustrate the similarity among SDT systems in how they modify programs.

2.1.1 Strata: A SDT Infrastructure

Strata is a reconfigurable and retargetable software dynamic translation infrastructure that is used as an experimental test-bed in this dissertation [79]. Different SDT systems such as a dynamic optimizer, a dynamic instrumenter and a software security checker, have been built using Strata [32,80,81]. Indeed, a SDT system is essentially a *client* that uses services provided by Strata. Strata is easily reconfigured for different purposes (clients). Its clients are representative of other SDT systems. For example, Strata's dynamic optimizer is similar to Dynamo [5], Mojo [14] and DynamoRIO [9]; Strata's dynamic instrumenter [49] is similar to Dyninst [62] and Pin [59].

Strata is organized as a virtual machine (see Figure 4). The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are held in a Strata-managed code cache. The Strata VM is entered by capturing and saving the application context (e.g., PC, condition codes, registers, etc.). Following context capture, the VM processes the next



(a)

(b)

Figure 4: Strata Architecture

application instruction. If a translation for this instruction has been cached, a context switch restores the application context and begins executing cached instructions on the host CPU.

If there is no cached translation for the next application instruction, the Strata VM allocates storage for a new *fragment* of translated instructions. A fragment is a straightline sequence of instructions[1]. The Strata VM builds the fragment by fetching, decoding, and translating application instructions one-by-one until an end-of-fragment condition is met. An end-of-fragment condition could be a loop back-edge or a branch instruction. When the end-of-fragment condition is met, a *context switch* restores the application context and the newly translated fragment is executed. In this way, the application is executed solely from the code cache. Specific details of what constitutes an end-of-fragment condition and how a context switch is performed are detailed in Scott et al. [81].

Figure 4(b) shows the components of the Strata VM. The Strata virtual machine is implemented as a set of target-independent *common services*, a set of *target-specific functions*, and a reconfigurable *target interface* through which the machine-independent and machine dependent components communicate. Implementing a new software dynamic translator often requires only a small amount of coding and a simple reconfiguration of the target interface. Even when the implementation is more involved (e.g., when

[1] Recently, the definition of Strata's fragment has been expanded to include more complex control-flow structures [37]. This thesis uses an earlier incarnation of Strata where fragments were single-entry, single-exit entities [81].

Untranslated Code	Translated Code
`0x1bc8 ld  [%o2+408],%o4`	`0x100c8 ld  [%o2+408],%o4`
`0x1bcc clr  %o3`	`0x100cc clr  %o3`
`0x1bd0 sll  %o3, 2, %g1`	`0x100d0 sll  %o3, 2, %g1`
`0x1bd4 ld  [%o2+%g1],%o5`	`0x100d4 ld  [%o2+%g1],%o5`
`0x1bd8 inc  %o3`	`0x100d8 inc  %o3`
`0x1bdc cmp  %o3, 0xff`	`0x100dc cmp  %o3, 0xff`
`0x1be0 ble  0x1bd0`	`0x100e0 ble  0x100f8`
`0x1be4 add  %o4,%o5,%o4`	`0x100e4 add  %o0,-16,%o0`
`...`	`... // Trampoline calling Strata`
`...`	`... // to translate code at 0x1be8`
	`0x100f8`
	`... // Trampoline calling Strata`
	`... // to translate code at 0x1bd0`

(a) (b)

Figure 5: SPARC code snippet that constitutes a Fragment and its translated counterpart

retargeting the VM to a new platform), the programmer is only obligated to implement the target-specific functions required by the target interface; common services do not have to be re-implemented or modified.

Figure 5(a) uses an example binary code snippet from a SPEC2000 benchmark to illustrate dynamic translation with Strata. The code snippet contains a single branch instruction and constitutes a *fragment*. Figure 5(b) shows the corresponding fragment generated by Strata. Note that the binary locations for each instruction have changed from 0x1b** in Figure 5(a) to 0x100** in Figure 5(b). The addresses in Figure 5(a) are application binary locations in the text segment of the executable, and the addresses in Figure 5(b) are locations in the code cache. Also note that the target of the branch instruction has been modified during translation and that no other instructions were changed. The branch targets have been modified to point to *trampoline* code (e.g., the new target of the branch instruction at `0x100e0` points to the trampoline at `0x100f8`). Trampolines are code sequences that save execution context and transfer control to Strata when previously untranslated code is encountered.

2.1.2 Overhead Reduction in SDT

The overhead of monitoring and modifying a running program's instructions can be substantial in SDT systems. This is unfortunate because SDT has numerous advantages in modern computing environments and interesting applications of SDT continue to emerge. There are a number of well-known techniques to reduce dynamic translation overhead. This section describes three overhead reduction techniques. Most SDT systems use some or all of these techniques [3,5,9,11,14,18,20,64,80,85,95,96].

(a) Fragment Linking

In a typical SDT system's basic mode of operation, each control transfer instruction (branch instruction) is replaced with a code trampoline that returns control back to the SDT system. The SDT system then translates instructions at the target of the branch. On a context switch from application code (in the code cache) to the SDT system, a search is performed (e.g., using a hashtable) to determine if there is a cached fragment corresponding to the current PC. If a cached fragment is found, the SDT system immediately switches back to the application in the code cache; otherwise, the SDT system builds a corresponding fragment before context switching back. The context switches comprise the majority of SDT overhead [48].

A large portion of the context switches due to non-indirect branches can be eliminated by "linking" fragments together as they materialize into the code cache. For instance, when one or both of the destinations of a PC-relative conditional branch materialize in the code cache, the corresponding trampoline can be rewritten to transfer control directly to the appropriate code cache locations rather than performing a context switch and control transfer to the SDT system. Fragment linking results in significant performance improvements. For example, fragment linking in Strata improves the performance of SPEC2000 benchmarks by over a factor of 4 [48].

(b) Indirect Branch Translation Cache

After applying fragment linking, the remaining overhead is primarily due to indirect control transfers [48]. Because the target of an indirect branch is only known when the branch executes, the SDT system cannot link fragments ending in indirect control transfers to their targets. As a consequence, such fragments must save the application context and call the SDT system with the computed branch-target address. It is likely that the requested branch target is already in the code cache, so the SDT system can simply restore the application context and execute the target fragment. However, for programs that execute large numbers of indirect control transfer instructions, the overhead of handling the indirect branches can still be substantial [79].

To overcome the overheads associated with handling indirect branches, a common technique is to use an Indirect Branch Translation Cache (IBTC). An IBTC is a cache of targets of indirect branches. During translation of an indirect branch, the SDT system generates code that performs a lookup in the IBTC to determine if the cached target is the designated target; if so, control is transferred to the cached target. If the target is not already cached, a new entry is created in the IBTC. There are different policies for IBTC maintenance: a cache for each indirect branch, a cache for all indirect branches, or a combination thereof, are some of the possibilities. While IBTC is used by a number of SDT systems, other approaches to reduce indirect branch overheads exist. A comprehensive analysis of different indirect branch handling policies is presented in a recent work [36]. Indirect branch handling and fragment linking together to reduce the SDT overhead in Strata to an average of 4 % [37].

(c) Instruction Traces

Another common technique for improving performance of SDT systems is instruction traces. An instruction trace is a sequence of instructions on a frequently executed program path. Consider the example CFG in Figure 6(a) that contains basic blocks A, B, C, D and E. Assume that the path through blocks A, C, D and E is frequently executed. An instruction trace can be formed as shown in Figure 6(b) along the frequently executed path.

Instruction traces provide instruction cache locality benefits because frequently executed code is spatially close. In addition, instruction traces also aid in reducing the cost of indirect branch handling [48]. One technique to form instruction traces involves inserting counters at loop back edges in the code cache. When a counter reaches a threshold, the loop is assumed to be frequently executed and an instruction trace is constructed using the subsequent path through the loop body. This technique is used in Dynamo[5], DynamoRIO [9] and Strata-DO[32]. In addition to helping reduce SDT overheads, the instruction traces

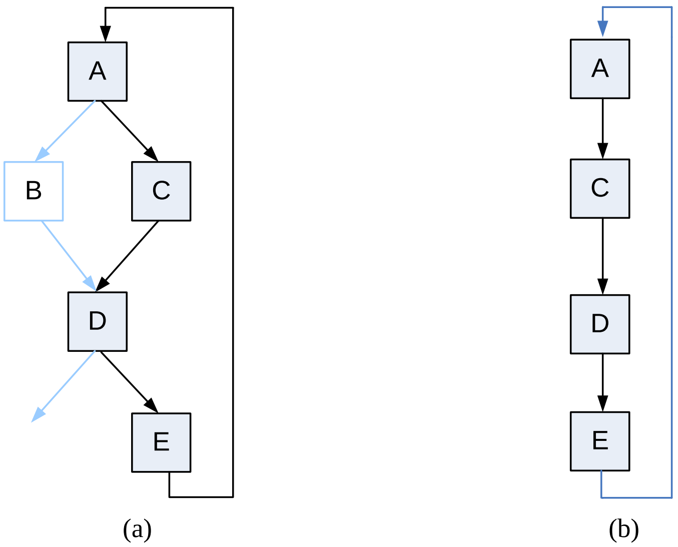


Figure 6: A frequently executed path (e.g., A-C-D-E in (a) above) can form a trace shown in (b)

are excellent candidates for dynamic optimization. Traces have simple control flow that leads to fast analyses suitable in a dynamic setting. Indeed, several dynamic optimizers use instruction traces as the granularity at which they apply optimization [5,9,14,32].

2.1.3 SDT Systems

A SDT system can be realized as a client of Strata by overloading its translate phase (see Section 4(a)) for a specific purpose. For example, each instruction can be translated to a different instruction set architecture. In this way, a binary translator can be realized. This section describes four diverse SDT systems, including a dynamic optimizer, a dynamic instrumenter, a software security checker and a binary translator. This section does not describe the specific code modifications made by different SDT systems, rather it gives an overview of the functionalities of different SDT systems.

(a) Dynamic Optimizer

A dynamic optimizer applies optimizations based on runtime program behavior. Dynamic optimization is useful because programming paradigms such as JIT compilation, object-oriented and aspect-oriented programming limit the efficacy of traditional optimizations. Further, in a SDT system, dynamic optimization can mitigate the overheads imposed by dynamic translation. Examples of dynamic optimizers include JIT-based systems (e.g., Java HotSpot [90] and .Net CLR [45]) and SDT-based systems (e.g., Dynamo [5], DynamoRIO [9], Mojo [14], ADORE [57] and Strata-DO [32]).

A dynamic optimizer can be built as a SDT client in which a program is optimized during translation. As a program executes from the code cache, the SDT system tracks code regions where optimizations may be beneficial. Optimizations are applied and code is generated in the code cache. To build a dynamic optimizer using Strata, the translate phase in Figure 4(a) is overloaded to apply optimizations in addition to regular translation.

Untranslated Code	Dynamically Optimized Code
0x1bc8 ld [%o2+408],%o4 **0x1bcc clr %o3** 0x1bd0 sll %o3, 2, %g1 0x1bd4 ld [%o2+%g1],%o5 0x1bd8 inc %o3 0x1bdc cmp %o3, 0xff 0x1be0 ble 0x1bd0 0x1be4 add %o4,%o5,%o4	0x100c8 ld [%o2+408],%o4 0x100cc sll %o3, 2, **%o5** 0x100d0 ld [%o2+**%o5**],**%o1** 0x100d4 inc %o3 0x100d8 cmp %o3, 0xff 0x100dc ble 0x100f4 0x100e0 add %o0,-16,%o0 ... // Trampoline calling DynOpt ... // to translate code at 0x1be8 0x100f8 ... // Trampoline calling DynOpt ... // to translate code at 0x1bd0

(a) (b)

Figure 7: Optimizations can be applied during the *translate* phase of SDT

Figure 7 uses the SPARC code snippet from Figure 5 to illustrate dynamic optimization. Figure 7(a) shows the untranslated code snippet, and Figure 7(b) shows the dynamically optimized counterpart. The instruction at location 0x1bcc in the untranslated code is eliminated by dynamic optimization. Also, the allocation of registers is different in the dynamically optimized code than the untranslated code (see instructions at locations 0x100cc and 0x100d0 in optimized code). Finally, the generated code is identical to that in Figure 5(b) (including the trampolines), except optimizations have been applied.

(b) Dynamic Instrumenter

Dynamic instrumentation is a technique by which *external code* is inserted into an executing program. External code is not part of the original application and is used for monitoring or controlling the behavior of an application. For example, Dyninst allows external profiler code to be attached to programs [62]. In addition to profilers, dynamic instrumenters (e.g., Dyninst [62], FIST[49] and Pin[59) can be used to build software security tools [47], testing tools [63,93] and debuggers [99].

Untranslated Code	Dynamically Instrumented Code
0x1bc8 ld [%o2+408],%o4	**0x100c8 save %sp, -96, %sp**
0x1bcc clr %o3	**0x100cc sethi %HI(ctr),%o1**
0x1bd0 sll %o3, 2, %g1	**0x100d0 ori %o1,%LO(ctr),%o1**
0x1bd4 ld [%o2+%g1],%o5	**0x100d4 ld %o2, [o1]**
0x1bd8 inc %o3	**0x100d8 add %o2, 1, %o2**
0x1bdc cmp %o3, 0xff	**0x100dc sd %o2, [o1]**
0x1be0 ble 0x1bd0	**0x100e0 restore**
0x1be4 add %o4,%o5,%o4	0x100e4 ld [%o2+408],%o4
...	0x100e8 clr %o3
...	0x100ec sll %o3, 2, %g1
	0x100f0 ld [%o2+%g1],%o5
	...

(a) (b)

Figure 8: Instrumentation of a counter into a fragment via Dynamic Instrumentation

Dynamic instrumenters use SDT as the underlying technique. To build a dynamic instrumenter as a client of Strata, the translate phase in Figure 4(a) is overloaded to "stitch" external code into the translated code in the code cache. In the translate phase, the translator looks for certain program properties to be satisfied for external code to be inserted. For example, a *basic block counter*[1] looks for the start of a basic block and inserts code to increment a counter at that point.

Figure 8 illustrates dynamic instrumentation using the SPARC code snippet from Figure 5. The untranslated code is shown in Figure 8(a) and dynamically instrumented code in Figure 8(b). The first seven instructions have been inserted for instrumentation in Figure 8(b). There is no counterpart to these instructions in the untranslated code (compare Figure 8(a) and Figure 8(b)). The extra instructions are the

[1] A basic block counter is an instrumentation application that counts the number of basic blocks executed during a program run.

external code that increment a *counter* every time the fragment is executed. The external code consists of a prologue to save the context of execution (*save* instruction) and an epilogue to restore the context (*restore* instruction). The prologue and epilogue provide a new context of execution for the external code. The instructions between the prologue and epilogue increment a counter (`ctr` in Figure 8(b)). The instructions following the epilogue are the usual translated code from Figure 5(b).

(c) Binary Translator

A binary translator translates a program binary from one instruction set architecture to another. Binary translation helps overcome the barriers to entry associated with the introduction of a new OS or CPU architecture. For example, Rosetta uses binary translation to allow legacy PowerPC applications on Intel processors available in newer generation of Apple hardware [2]. Similarly, Transmeta's Code Morphing technology was used to allow unmodified Intel IA-32 binaries to run on the low-power VLIW Crusoe processor [25]. The FX!32 dynamically translates x86 binaries to run on Alpha processors [15].

A binary translator can be implemented using Strata by overloading the translate phase of Strata VM in Figure 4(a) to perform binary translation. Although it may be easy to translate some sequence of instructions from one machine target to another (e.g., ALU and logical instructions), serious differences between machine targets can make other translations much more difficult. For instance, differences in calling conventions, memory layouts and endianness, and exception behavior can make straightforward translation difficult or impossible. In such cases, binary translators generate sequences of instructions that emulate the source machine behavior on the target machine.

Figure 9 illustrates binary translation of SPARC code snippet into x86 (see Figure 9(a) and Figure 9(b)). The translated x86 code performs the same functions as the translated code in Figure 5(b). Note that the dynamic translator still generates trampolines at the end of x86 fragment (see Figure 9(b)), so that control is transferred back to the SDT system for binary translation of the next fragment.

Untranslated Code	Binary Translated Code
`0x1bc8 ld  [%o2+408],%o4`	`0x848790 mov $0x804b4a0,%edx`
`0x1bcc clr  %o3`	`0x848795 lea 0x0(%esi),%esi`
`0x1bd0 sll  %o3, 2, %g1`	`0x848798 mov (%edx,%ebx,4),%eax`
`0x1bd4 ld  [%o2+%g1],%o5`	`0x84879b inc %ebx`
`0x1bd8 inc  %o3`	`0x84879c add %eax,0x804b8a8`
`0x1bdc cmp  %o3, 0xff`	`0x8487a2 cmp $0xff,%ebx`
`0x1be0 ble  0x1bd0`	`0x8487a8 jle 0x8048798`
`0x1be4 add  %o4,%o5,%o4`	`// Trampolines`
`...`	`...`
`...`	

(a) (b)

Figure 9: Binary Translation from SPARC to x86

(d) Software Security Checker

A software security checker verifies whether malicious code, such as a virus or a worm, has infected a program or whether the program has been maliciously modified otherwise [47,80]. There are numerous ways to build software security checkers and often the term can be used to mean entirely different tools. In the context of SDT, a software security checker refers to a system that monitors the execution of a program and verifies that it has not been maliciously affected.

Several software security systems have been proposed including those that perform sandboxing, program shepherding and instruction set randomization [44,47]. Sandboxing is a technique to instrument sensitive code regions in a program so that certain checks can be performed before executing those code regions [13]. Program shepherding is a technique that enforces certain security policies on code during dynamic translation and during execution [47]. Program shepherding makes use of sandboxing. Instruction set randomization is a technique to encrypt a program using a key such that the encoding for each instruction appears to be completely random [44]. The program is decrypted during dynamic translation and then executed from the code cache. If malicious code is injected into the encrypted program, it would be decrypted by the dynamic translator and stored in code cache before execution. Since the malicious code would not have been encrypted in the first place (the encryption key is secret), the program would likely crash while executing the “decrypted” malicious code. Hence, the program is secured from malicious modifications.

2.2 Source Level Debugging

Source level debuggers present a source view of a program even though the binary format of the program actually executes on hardware. To provide the source view, debuggers traditionally use static symbol information associated with a program to relate source code to the binary code. The static symbol information, also called *debug information*, is generated during compilation by the compiler that produced the binary code. Debug information is saved as part of the executable program. When a debugger is invoked, it gains control of the program via services provided by the operating system [33,60]. The operating system services permit the debugger to control and inspect the state of the executing program.

From a user's point of view, source level debugging involves inserting a breakpoint at a source location (e.g., a source line number or a method name) and inspecting variable values when execution pauses at the breakpoint. Debuggers provide many other facilities, including printing the execution stack trace, single-stepping through execution, displaying source code to the screen, displaying variable values in user friendly manner (e.g., a tree of values), among others. These facilities can be built using the basic functionality of breakpoints and value inspection[1]. Indeed, operating systems services, such as `ptrace` or `/proc`, are limited to *reading* and *writing* of breakpoints and values [33,60].

A debugger's task is difficult when dealing with dynamically translated programs due to three challenges discussed in Section 1.2. This section discusses prior work on these challenges for debugging statically generated code and a discussion of why they are insufficient for SDT systems.

2.2.1 Code Transformation

Debuggers for optimized code have to deal with program transformations made by optimizers. When a program is optimized, the code is transformed to perform the same computation more efficiently. Code

[1] Certain debug facilities cannot be entirely built upon these basic facilities. Consider detecting race conditions in concurrent programs, for example. This thesis does not target those facilities.

transformation renders debug information inconsistent with the binary code. For example, a computation can be eliminated or moved earlier or later than in the unoptimized code. This leads to both the code location and data value problems. Therefore, techniques are needed to hide the effect of code transformations from a debug user. A substantial amount of research work has been performed to permit source level debugging of optimized code. Based upon how the approaches tackle the debugging challenges associated with code transformation, they can be categorized as follows.

(a) Avoidance

The first approach avoids debugging transformed code. For example, avoiding transformations that cause problems to debugging. Fritzon's debugging system limited optimizations to within a source statement [29]. Pollock and Soffa permit all optimizations except those that affect debugging requests [72]. The debugging requests must be provided before program execution begins so that necessary compiler optimizations can be inhibited. Holzle, Chambers and Ungar provide another enhancement to Pollock and Soffa's techniques by dynamically "de-optimizing" selected parts of programs that affect debugging requests [38]. The Java HotSpot compiler uses the dynamic deoptimization approach [90]. The Common Language Runtime from the .NET framework does not allow debugging when optimizations are enabled [70].

A completely different approach to avoiding debugging problems due to code transformation was taken by Brooks, Hansen and Simmons by exposing the effects of optimization. They highlighted statements to visually communicate program transformations [8]. Similarly, Tice and Graham's *Optview* generates source code corresponding to optimized code and thus conveys the effects of optimization to a user [91]. A debugger, *Optdbx*, uses the source code generated by Optview to perform debug queries based on the optimized source program.

(b) Inference

Program transformations can expose latent bugs in software applications [34]. Avoiding program transformations, therefore, also avoids these latent bugs. The avoidance approach, therefore, is insufficient for accurately debugging transformed programs. Perhaps more importantly, debugging and testing code that is actually deployed is vital from a software engineering standpoint.

Another approach to debugging transformed code involves automatically inferring the effect of code transformations and then hiding these effects in a debug session. Copperman [22] and Wismuller [100] use data-flow analysis to determine which variables are current. The code location problem is handled by mapping source code to optimized code after all optimizations have been performed. Their approaches are similar, but Wismuller's approach is more general. These works are very interesting because they do not require modification to the optimizer. The effects of optimization are automatically determined through data flow analysis of optimized and unoptimized code.

(c) Propagation

One disadvantage of the inference approach is that the data flow analyses involved is very complex and, as a result, these techniques have not been implemented. The third approach to debugging optimized code is to modify the optimizer and determine code transformations and propagate this information as code is further modified through various optimization passes. The first work on debugging optimized code was done by Hennessy, and it used the propagation approach [34]. Hennessy recognized that program transformation can lead to values computed earlier or later in the optimized program than in the unoptimized program. A technique to determine which values can be reported correctly to the user was proposed. For values that could not be reported correctly (due to transformation), techniques to "recover" those values were also proposed. Hennessy dealt with only local optimizations (within a basic block) and did not handle aliasing of variable values.

Zellweger's work focused on accurately mapping a source statement to each of the corresponding object code locations [105]. A limited number of code transformations are handled in this work. Zellweger does not solve data value problems.

Adl-Tabatabai's work, using data-flow analysis to find current variables, is similar to Copperman's and Wismuller [1,2]. However, Adl-Tabatabai makes a simplifying assumption that optimizations cannot transform code arbitrarily, that is, a computation cannot be moved into a path where it did not exist before. In contrast to using a summary effect of optimizations, as Copperman and Wismuller do, Adl-Tabatabai propagates the effect of each optimization step through all the optimization phases.

A limitation of Adl-Tabatabai's work is that it cannot correctly determine variable values that are dependent on runtime paths. For example, if a variable's value at a program location depends on which path was taken to reach that location, Adl-Tabatabai's approach cannot yield the correct answer. Several works on debugging optimized code make use of runtime information to overcome this limitation. Dhamdhere et al. developed a dynamic currency determination technique in which a minimal unrolled graph of the program is constructed and basic blocks are time-stamped during execution [27]. In this way, a partial history of execution path is determined, and the history is used to precisely determine variables that have data value problems. Dhamdhere et al. do not completely solve the data value problem and do not consider the code location problem.

Wu et al. selectively emulate portions of program near breakpoint locations in the unoptimized order of statements [102]. Debug queries are performed using values computed during emulation when the values are otherwise not reportable due to data value problems. There are several limitations with this approach. For example, selective emulation can be very time consuming when a statement is moved across a function call due to code transformation. Further, path sensitive data values are not correctly reported.

More recently, the Fulldoc debugger by Jaramillo et al. uses data flow analysis to compare unoptimized code with optimized code [41,42]. Variables with the data value problem are determined at each potential breakpoint location. In a debug session, runtime values of variables that are computed earlier are gathered using invisible breakpoints. Values that are computed later than in the unoptimized program are gathered by the technique of record-replay [75]. The runtime values are saved in a value pool and the correct value is displayed for a given variable according to the static information collected via data flow analysis. Fulldoc is able to report every value that is computed in the optimized program. Since Fulldoc's techniques report the most values, this thesis builds upon Fulldoc's approach to handle some code transformations, in particular movement of instructions (see Section 5.2 for more details).

Prior Work and this Dissertation. The first approach to debugging transformed (optimized) code avoids code transformation. One problem using this approach is the potential of missing latent bugs that are exposed due to code transformation. This problem is equally relevant in SDT systems. Additionally, there may not be a way to execute a program without dynamic translation. For example, a software security system may be embedded in the operating system such that it is impossible for developers to turn off the security system. As a result, debugging transformed code may be unavoidable.

The second approach to debugging optimized code treats optimizations as a black box and determines the effect of all the code transformations that took place. The problem with this approach is that its computational complexity is high. Code transformations in SDT systems can be different than those performed by static optimizers. For example, SDT systems can insert additional code into a program such that the additional code is interspersed throughout the program. Using the black box approach would be difficult in this case because there is no way to determine which instructions in the transformed code should be related with the original code.

The third approach to debugging optimized code is to propagate information about code transformation as they are applied. Debug information is generated and composed as optimizations are performed. One problem with each of the prior works using the propagation approach is that they all handle only code transformations resulting from optimizations. SDT systems can perform new kinds of program transformations such as insertion of additional unrelated code. Further, SDT systems can transform code that was previously generated, transformed and executed. In the traditional optimizers, the debug information is generated after all optimizations have been applied. In SDT systems, the debug information must be generated after one set of transformations is applied and before this transformed code is executed. The debug information can then be composed with (or regenerated) after the next set of transformations. The difficulty in composing or regenerating debug information is that after transformations are applied, the untransformed code is elided.

The solution is to preserve enough information about the untransformed code so that the transformed code can be related with the original program at any time. SDT systems apply code transformations at a granularity that is not statically fixed (e.g., an execution path as opposed to a method). At different levels of optimization, the granularity can change and instructions can move across code regions. For static optimizers, this is akin to inter-procedural optimization where each instruction must be related back to its original procedure. Prior work does not handle inter-procedural optimizations.

Finally, the approaches taken by the prior work would require building a debugger for each SDT system. This thesis aims to provide a general solution that can be used by any SDT system. Code transformations handled by the proposed debugger should encompass those performed by diverse SDT systems, yet be independent of an individual SDT system. Therefore, the construction of debug information should be abstracted away from the SDT system. Prior work does not provide such techniques.

2.2.2 Online Communication

Debuggers for just-in-time (JIT) compiled code need online communication with the JIT compiler. In a typical JIT compiler, methods are compiled at load time [45,90]. The debug interface provided by Java allows communication between the Java virtual machine (JVM) and a debugger using the Java Platform Debug Architecture (JDPA) [43]. This interface permits the debugger to send commands and queries to the JVM. The JVM acts as black box to the debugger and is responsible for not only computing the debug information but also for dispatching debug actions on behalf of the debugger. The JDPA presents a monolithic debugging architecture where debug information is generated and used by the JVM. In some sense, the wire protocol provided to the outside debugger is similar to remote debugging capabilities of Gdb [88], where the debugging actions are dispatched by another entity on behalf of the debugger. The debug architecture for .NET is similar [70].

Conceivably, debug information that is generated for statically compiled programs can be generated at runtime for dynamically translated programs and stored along with the binary program on disk. This approach is used by Pauw et al. for debugging object oriented programs [69].

Prior Work and this Dissertation. While online communication for debugging purposes is possible in JIT compilers and remote debuggers, the debugging architecture in these systems is monolithic. The goal of this thesis is to provide techniques that can be used by multiple SDT systems. With the approach of JIT compilers and remote debuggers, each SDT system would have to develop their own debugging framework. This thesis strives to develop lightweight mechanisms that can provide debugging support to a wide array of SDT systems and efficient communication mechanisms between a SDT system and a debugger.

2.2.3 Efficiency

Efficiency of generating debug information is a bonus for systems that generate debug information statically: it is not a requirement. Efficiency is important, but not critical for JIT compiled programs either. JIT compilers generate code at a coarse granularity (e.g., methods) and are invoked (for code generation) rather infrequently [3,45,90]. Therefore, the overhead of generating debug information is not substantial. Some debuggers for JIT compiled code use an interpretation mode where the overhead of interpretation is substantial when compared to the overhead of generating debug information. Prior work on debugging has not focused on efficiency as a prime concern.

Prior Work and this Dissertation. The cost of generating debug information can be substantial in SDT systems due to several reasons. First, SDT systems typically generate and transform code at a fine granularity, such as basic blocks. Therefore, debug information needs to be generated and communicated frequently, necessiating efficient representations and communication mechanisms. Second, SDT systems often have extensive code duplication. The code cache sizes of SDT systems can be much larger than the text segments of the untranslated programs. DynamoRIO, for example, keeps the size of its code cache unbounded for this reason [11]. Debug information must be generated whenever new code is generated. Finally, some SDT systems periodically flush the code cache eliminating the entire code generated so far [5,10,81]. Other systems can flush smaller portions of the code cache [26]. With flushes, the debug information also needs to be removed. Consequently, when the same code is later generated, debug information also needs to be generated.

The problem with large amounts of debug information is that the cost of generating them adds to the overall runtime of the translated program. To permit post-mortem debugging and to allow debugging of programs whose execution has begun outside of the debug session, it is vital that debug information is generated even outside of debug sessions. Therefore, the overheads associated with generation of debug information must be minimized. Efficiency was not as vital for any of the prior works as for this dissertation. Indeed, slowdowns resulting from generation of debug information has not even been reported in most prior works. In terms of complexity, the prior works on debugging optimized code have a non-linear complexity for generation of debug information (most prior works use iterative data flow analyses). This thesis aims to achieve debugging goals with a linear complexity.

2.3 Summary

This chapter used the Strata infrastructure [79] to describe how a basic SDT system works. Strata provides a client model of SDT systems, where new SDT systems can be realized as a client of a base infrastructure. This chapter presented four example SDT systems based on this model to illustrate the capabilities of SDT systems. This chapter also discussed prior works on source level debugging that are related to this research. Finally, this chapter discussed why the prior research on source level debugging is insufficient in addressing the debugging challenges posed by SDT systems.

Chapter 3. Tdb: A Debug Framework

This dissertation presents debugging techniques for SDT systems that allow: (1) program transformations to be tracked throughout the lifetime of a program; (2) debug information to be generated simultaneously with program transformations; and (3) the debug information to be immediately available to a debugger. The debugging techniques are encapsulated in a new framework, called Tdb.

Tdb can be used to implement a source level debugger for a given SDT system. This chapter gives an overview of the Tdb framework. The next section, Section 3.1, describes the organization of Tdb and shows a high-level view of how debugging is performed with it. Section 3.2 describes how program transformations are tracked in Tdb. Section 3.3 shows how debug information is generated and used in Tdb. Section 3.4 gives an example to illustrate debugging with Tdb. Finally, Section 3.5 summarizes the chapter.

3.1 Debugging with Tdb

The Tdb framework consists of three components: a *SDT system*, a *debug engine*, and an existing source level debugger, called the *native debugger*. The organization of Tdb is shown in Figure 10. In Tdb, the SDT system is modified by adding a component, called the Program Tracker, which determines programmatic modifications made by the SDT system. The debug engine is a component that computes debug information based upon the program tracker's output. This debug information is used by the native debugger to hide the effects of program transformations and the presence of the SDT system. The native debug-

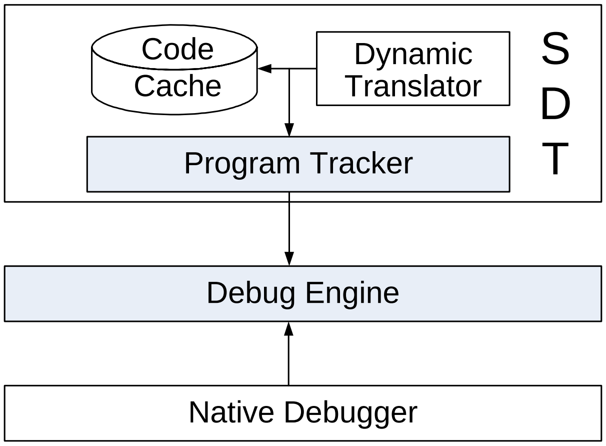


Figure 10: The Tdb Framework

ger is an existing source level debugger (e.g., Gdb [88] or Dbx [56]) that is modified to communicate with the debug engine.

With Tdb, debugging is a three-step process, as illustrated in Figure 11. In the first step, the program tracker generates information about the code modifications performed by the SDT system. The modification to each instruction and data value are tracked and represented as attributes that are associated with those individual instructions/data values. The attributes, called Transformation Descriptors, represent the effect of all transformations applied to a given instruction/data value. As a result, each instruction and data value that is transformed by the SDT system has one transformation descriptor associated with it. Note that the transformation descriptors only capture how a program is modified — they do not specify how a debugger should hide the program modifications from a debug user.

In the second step, the transformation descriptors are used by the debug engine to generate debug information. Debug information can be used by the native debugger to hide the effect of program transformations. For example, if a transformation descriptor specifies that a certain data value has been eliminated during dynamic translation, the corresponding debug information will specify how to determine the deleted value in a debug session.

The final step of Tdb is the use of debug information by a debugger. Tdb requires modifications to an existing debugger[1] so that its actions on a program are instead targeted to the debug engine. The debug

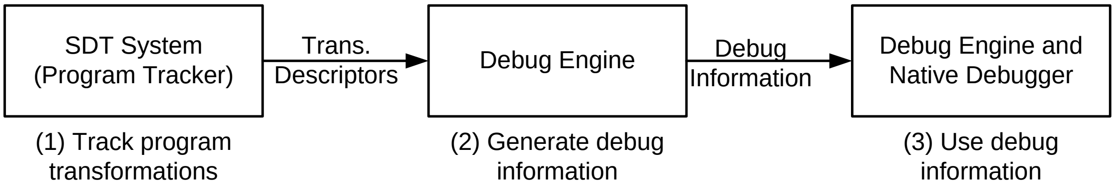


Figure 11: Three step debugging

engine in turn, performs the same actions on the translated program. As an example, instead of inserting a breakpoint (e.g., writing a breakpoint trap instruction) into the program binary, the native debugger invokes the debug engine. The debug engine finds out an appropriate code cache location for the breakpoint using debug information and inserts the breakpoint in the code cache. Recall that a program is always executed from the code cache in SDT systems, therefore breakpoints inserted in program binary will never be hit. With Tdb, the breakpoints are inserted at appropriate locations in code cache and are hit in an expected manner.

In Tdb, the first two steps are performed continuously during a program's execution as new code is generated or existing code is modified by the SDT system. The third step is performed on-demand in response to commands and queries of a debug user.

3.2 Tracking Program Transformations

When SDT systems translate code, they may do so in passes, similar to traditional optimization passes. Once all passes have been applied, binary code is generated and emitted into the code cache. The program tracker generates fine-grained information, represented as transformation descriptors, for each instruction that was translated and each transformed data value. The transformation descriptors consist of information to identify each instruction/data value and to capture the effect of transformations applied. For example, the following triple represents a transformation descriptor representing code movement.

```
Transformation Descriptor(insn): <CMove, orig, new>
```

[1] Section 6.3.1 describes a technique by which actions of existing debuggers can be intercepted, thus avoiding any modification (and recompilation) of the debugger.

The triple indicates the type of descriptor, i.e., `CMove` for code movement, and specifies the original and the new locations for the instruction. The information captured by this descriptor is enough to identify the instruction (via its original program binary location, `orig`) and describe the effect of code transformations applied to it (i.e., code movement from `orig` to `new`).

Importantly, the transformation descriptors do not capture what transformations were applied. For example, if a dynamic optimizer applies a set of optimization passes that result in an instruction being moved from its original neighbors, the transformation descriptor for that instruction will simply indicate that code movement took place.

While SDT systems may perform semantically different program transformations (e.g., optimization vs. binary translation), each program transformation can be described as a set of basic code edits, including insertion, deletion and movement of code and data values. The transformation descriptors capture these basic edits and, therefore, are applicable regardless of the purpose of dynamic translation. An advantage of transformation descriptors is that, due to their fine-grained nature and lack of semantic information (e.g., what set of optimizations resulted in an instruction eventually being deleted), they can be quickly generated and consumed. Further, the transformation descriptors make the generation and use of debug information independent from the SDT system. This property enables portability for the techniques developed in this research.

The implementation of the program tracker is usually straight-forward. In one implementation of the Tdb framework for Pin [59], the program tracker consisted of less than ten lines of code [52]. Two other implementations of the Tdb framework are discussed in Chapters 7 and 8 respectively. In both implementations, the program tracker was straight-forward to implement. The next chapter, Chapter 4, provides a detailed description of the different kinds of transformation descriptors available.

3.3 Generation and Use of Debug Information

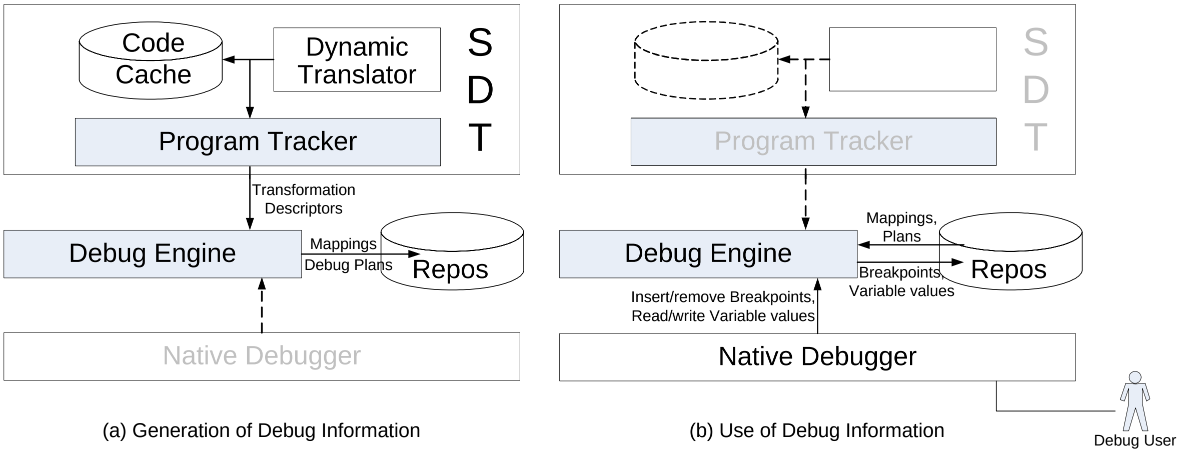


Figure 12: Functionalities of the Debug Engine

The debug engine is the central component in Tdb, which facilitates debugging of dynamically translated programs. The debug engine performs two distinct functionalities. First, it consumes transformation descriptors produced by the program tracker and generates debug information. Second, when invoked by the native debugger, it uses the debug information to facilitate source level debugging. Overall, the debug engine essentially extends the capabilities of existing source level debuggers to support SDT systems. In the Tdb framework, the existing debuggers are still responsible for finding out the binary counterparts for the source level constructs used by debug users. The debug engine provides the extra step of relating untranslated code and data with the dynamically translated code and data values. The functionalities of the debug engine are depicted in Figure 12 and discussed below.

3.3.1 Generation of Debug information

The debug information generated by the debug engine consists of *mappings* and *debug plans*. A mapping relates two code or data value locations. Mappings are useful in determining code cache location where breakpoints should be inserted instead of the corresponding program binary locations. A debug plan guides extraction of runtime values in the program. Debug plans are used when expected variable values are not readily available because code transformations moved the computation to earlier or later in the program. In such cases, the debug engine uses the debug plans to determine variable values early enough for reporting purposes. The mappings and debug plans are generated from transformation descriptors.

Untranslated Code	Dynamically Optimized Code
`0x1bc8 ld  [%o2+408],%o4`	`0x100c8 ld  [%o2+408],%o4`
`0x1bcc clr  %o3`	`0x100cc sll  %o3, 2, %o5`
`0x1bd0 sll  %o3, 2, %g1`	`0x100d0 ld  [%o2+%o5],`**`%o1`**
`0x1bd4 ld  [%o2+%g1],`**`%o5`**	`0x100d4 inc  %o3`
`0x1bd8 inc  %o3`	`0x100d8 cmp  %o3, 0xff`
`0x1bdc cmp  %o3, 0xff`	**`0x100dc ble  0x100f4`**
`0x1be0 ble  0x1bd0`	`0x100e0 add  %o0,-16,%o0`
`0x1be4 add  %o4,%o5,%o4`	`... // Trampoline calling DynOpt`
`...`	`... // to translate code at 0x1be8`
`...`	`0x100f8`
	`... // Trampoline calling DynOpt`
	`... // to translate code at 0x1bd0`

(a) (b)

Figure 13: A breakpoint at location 0x1be0 in untranslated code leads to insertion of a breakpoint at location 0x100dc in dynamically optimized (translated) code; register `%o5` corresponds to `%o1`

3.3.2 Use of Debug Information

The debug engine is invoked when the native debugger takes an action on the binary program. These actions include the insertion and removal of breakpoints and a read or write of variable values. The debug engine uses debug information in its repository (see Figure 12) to take the same action on the translated code. Additionally, the debug engine saves runtime values and live breakpoint information in the repository. Chapter 5 describes the debug engine in detail.

3.4 Example Debugging Session with Tdb

To illustrate how debugging is done with Tdb, consider the example in Figure 13. Suppose a traditional debugger inserts a breakpoint at untranslated location `0x1be0` in response to a user placing a breakpoint at a corresponding source statement. In Tdb, the debug engine intercepts the action of the native debugger to insert a breakpoint. Subsequently, another breakpoint is inserted at the corresponding translated location

`0x100dc.` When execution is paused, assume that the value of a variable residing in register `%o5` is queried. The native debugger performs a read operation on register `%o5.` In Tdb, the debug engine intercepts this action and instead reports the value stored in register `%o1`. Note that the value in register `%o1` at `0x100dc` corresponds to the value in `%o5` at `0x1be0`. In this way, the program is debugged at the source level even though the underlying binary is transformed during execution.

3.5 Summary

This chapter presents an overview of the Tdb framework. It outlines two components of the framework: The program tracker and the debug engine. The program tracker generates information based upon transformations that a SDT system applies to a program. The debug engine is responsible for generating and using debug information for a dynamically translated program. Tdb requires minimal to no modifications to existing debuggers. Details of each component of Tdb and implementations of the framework are described in the following chapters.

Chapter 4. Transformation Descriptors

A transformation descriptor is an attribute of an instruction or a data value that represents the effect of all transformations applied to that instruction (or data value). Transformation descriptors are computed by the program tracker after all transformations have been applied to an instruction/data value in a pass. The descriptors are then communicated to the debug engine for generation of debug information.

Despite all the differences in the functionalities of SDT systems, the transformations performed by each SDT system can be viewed as a set of modifications to instructions and data values. Transformation descriptors capture these modifications. As a result, Tdb's use of transformation descriptors eliminates the differences between SDT systems (for expressing transformations) and provides portability across different SDT systems. In addition, since the transformation descriptors capture modifications to each instruction and data value in a program, every program transformation can be expressed using descriptors. Consequently, a debugger can hide the effect of each transformation from a user. Transformation descriptors, therefore, are a powerful and sufficient technique to describe program transformations performed by any SDT system.

There are six transformation descriptors that are applicable to instructions and two for data values. Table 1 summarizes the transformation descriptors. The descriptors for instructions describe insertion (CInsert), deletion (CDelete) and movement (CMove) of an instruction. In addition, there are two special descriptors: Identity and CFlush. The Identity descriptor is associated with each instruction that is translated but not modified by the SDT system. A CFlush descriptor signifies the elimination of an existing instruction from the code cache. There are two descriptors applicable to data values: DMove and DDelete.

DMove represents a change to the storage location of a data value. The DDelete descriptor signifies that a data value is no longer live at a program location. This chapter describes each of the transformation descriptors and illustrates the code transformations that generate them.

The rest of this chapter is organized as follows. The next section, Section 4.1, describes transformation descriptors applicable to instructions. Section 4.2 describes transformation descriptors for data values. Section 4.3 gives an example of how transformation descriptors would be generated for a transformation. Section 4.4 discusses how complex transformations can be described by means of the transformation descriptors. Finally, Section 4.5 summarizes the chapter.

Table 1: Summary of Transformation Descriptors

Transformation Descriptor	Summary
Identity <ID, Binary Location, Code Cache Location>	Indicates code relocation
CInsert <CI, NULL, Code Cache Location>	Instruction was not present in unoptimized code
CDelete <CD, Binary Location, NULL>	Instruction is deleted during optimization
CMove <CM, Binary Location, Code Cache Location>	Instruction was moved from its original location
CFlush <CF, NULL, Code Cache Location>	Instruction has been eliminated from code cache
DMove <DM, Code Cache Loc, OldLoc, NewLoc>	Storage location of data value has changed
DDelete <DD, Code Cache Location, VarLocation>	Data value is not available at program location

4.1 Code Descriptors

Code descriptors are transformation descriptors that describe modifications to instructions by program transformations. An individual instruction can be transformed in a limited number of ways. The transformations include basic transformations consisting of insertion, deletion and movement and complex transformations consisting of a combination of the basic transformations. A code descriptor describes each of the basic transformations. In addition, two other descriptors describe code relocation and code flush. Complex transformations can be described as a combination of basic transformations. Therefore, more complex code descriptors are not needed for complex transformations. Section 4.4 explains how complex instructions can be described in terms of the basic code descriptors.

In Tdb, a code descriptor is associated with each instruction that undergoes a transformation, including instructions that get eliminated during translation and instructions generated during translation. Each code descriptor is a triple consisting of a *type*, the binary *untranslated location* of an instruction and the *code cache location* of the instruction, as shown in Table 1. The discussion below illustrates code descriptors using examples, where each descriptor is only identified by its *type* for brevity.

Identity. An Identity descriptor denotes code relocation. While Identity does not indicate any transformation (except relocation), it distinguishes instructions that have been translated and are currently present in the code cache from those that are not present in the code cache. An Identity descriptor is shown in Figure 14. Note that the untranslated and translated code are the same in Figure 14; hence all translated instructions are associated with Identity. Identity does not exist for code that has not been translated.

Untranslated Code	Translated Code	Transformation Primitive
`0x1bc8 ld  [%o2+408],%o4`	`0x100c8 ld  [%o2+408],%o4`	Identity
`0x1bcc clr  %o3`	`0x100cc clr  %o3`	Identity
`0x1bd0 sll  %o3, 2, %g1`	`0x100d0 sll  %o3, 2, %g1`	Identity
`0x1be4 ld  [%o2+%g1],%o5`	`0x100d4 ld  [%o2+%g1],%o5`	Identity

Figure 14: Identity is used when code is NOT modified during dynamic translation

CInsert. The CInsert descriptor is used to describe insertion of an additional instruction. Code transformations frequently generate additional code. For example, a dynamic optimizer may apply the partial redundancy elimination optimization resulting in additional instructions. A dynamic instrumenter inserts “external” code for monitoring the application. Figure 15 uses a part of the code snippet in the example from Figure 8. A CInsert descriptor is used for each instrumented instruction. The original instructions appear after the instrumented code in the second column, for which Identity descriptors are generated.

Untranslated Code	Translated Code	Transformation Primitive
`ld  [%o2+408],%o4`	**`...`**	`...`
`clr  %o3`	**`add %o2, 1, %o2`**	`Code Insert`
`sll  %o3, 2, %g1`	**`sd  %o2, [o1]`**	`Code Insert`
`ld  [%o2+%g1],%o5`	**`restore`**	`Code Insert`
	`ld  [%o2+408],%o4`	`Identity`
	`clr  %o3`	`Identity`
	`sll  %o3, 2, %g1`	`Identity`

Figure 15: CInsert is used for add, store and restore (dynamically instrumented instructions)

CDelete. The CDelete descriptor describes the elimination of an instruction. CDelete is used when a code transformation results in a fewer number of instructions than the untranslated code. Figure 16 shows the use of CDelete when a binary translator eliminates an instruction, i.e., there is one fewer instruction than the untranslated code. In fact, the binary translator combined the last two instructions in the untranslated code into a single instruction. Section 4.4 explains how the program tracker generates CDelete when instructions are combined.

Untranslated Code	Translated Code	Transformation Primitive
`ld  [%o2+408],%o4`	`mov $0x804b4a0,%edx`	`Identity`
`clr  %o3`	`lea 0x0(%esi),%esi`	`Identity`
`sll  %o3, 2, %g1`	**`mov (%edx,%ebx,4),%eax`**	`Identity`
`ld  [%o2+%g1],%o5`		`Code Delete`

Figure 16: CDelete is used for the load instruction, which is considered deleted during optimization

CFlush. The CFlush descriptor describes the removal of an instruction from the SDT system's code cache. Instructions are removed when the code cache is full and space is needed to hold newly translated code. Instructions may also be removed for self-modifying code. CFlush is needed for consistency reasons: debug information (and information about live breakpoints) must be updated when existing code is removed from the code cache.

CMove. The CMove descriptor describes the movement of an instruction from its original location to an earlier or a later location. CMove is used when instructions are moved during dynamic translation. Figure 17 illustrates CMove. In the example, the bolded `ld` instruction is moved relative to its neighbors by means of a code scheduling optimization. The bolded `ld` instruction in the translated code has CMove.

Untranslated Code	Translated Code	Transformation Primitive
ld [%o2+408],%o4	ld [%o2+408],%o4	Identity
clr %o3	clr %o3	Identity
sll %o3, 2, %g1	sll %o3, 2, %g1	Identity
ld [%o2+%g1],%o5		
inc %o3	inc %o3	Identity
cmp %o3, 0xff	cmp %o3, 0xff	Identity
	ld [%o2+%g1],%o5	Code Movement

Figure 17: CMove is used when code scheduling optimization moves the load instruction

4.2 Data Descriptors

Data descriptors are transformation descriptors that describe modifications to data values or their storage locations. Similar to transformations affecting instructions, there are a limited number of ways in which data values can be modified by program transformations. There modifications are insertion, deletion and movement or a combination thereof. For debugging purposes, there are only two transformations of concern: elimination of a data values and a change in the location of a data value. A data descriptor describes each of these transformations, namely DDelete and DMove.

There are no data descriptors corresponding to the code descriptors CInsert, Identity and CFlush. A DInsert descriptor (corresponding to CInsert) would indicate computation of additional data values in the translated program. A program transformed by a SDT system usually performs additional computations. However, a debug user is not expected to know about these data values. In fact, the whole point of this thesis is to keep the debug user unaware of dynamic translation. Therefore DInsert is not needed. Identity is not needed for data values because any relocation of data values is already captured by DMove. Finally, DFlush (corresponding to CFlush) is not needed because SDT systems do not eliminate data values in a manner similar to flushing of the code cache. The only data descriptors used in Tdb, DDelete and DMove, are discussed below.

DDelete. The DDelete descriptor describes the absence of a data value at a program location. When an instruction that defines a variable is moved using CMove, a DDelete descriptor is associated with each instruction where a variable does not have an expected value. Figure 18 uses the example from Figure 17 that described the CMove descriptor. In Figure 18, code movement leads to unexpected values in register `%o5` at the `inc` instruction and the `cmp` instruction in the translated code (second column). Therefore, a DDelete descriptor is generated for these instructions in addition to the Identity descriptor.

Untranslated Code	Translated Code	Trans. Descriptor
`ld   [%o2+408],%o4`	`ld   [%o2+408],%o4`	`Identity`
`clr   %o3`	`clr   %o3`	`Identity`
`sll   %o3, 2, %g1`	`sll   %o3, 2, %g1`	`Identity`
`ld   [%o2+%g1],%o5`		
`inc   %o3`	`inc   %o3`	`Identity, DDelete`
`cmp   %o3, 0xff`	`cmp   %o3, 0xff`	`Identity, DDelete`
	`ld   [%o2+%g1],%o5`	`Code Movement`

Figure 18: DDelete is used when code scheduling optimization moves the load instruction

DDelete is used for the entire live range where one or more data values have been eliminated. Figure 19 shows a more general case. Figure 19(a) shows the movement of an instruction that defines the variable `x`, which overwrites a previous value of `x`. DDelete is used for the part of the live range that is overwritten due to the code movement, as shown by the curly brace in the figure. Similarly, if an instruction is moved later (shown in Figure 19(b)), the value of a variable can be computed later than in the original code. A DDelete descriptor is associated for the part of live range where this change appears (see the curly brace in Figure 19(b)). DDelete is also used when a code (or data) transformation eliminates a live range. The example in Figure 19(c) involves elimination of a definition of variable `x` resulting in removal of a live range of the variable. Clearly, there is no use of the variable in the live range (otherwise the transformation would likely have been invalid).

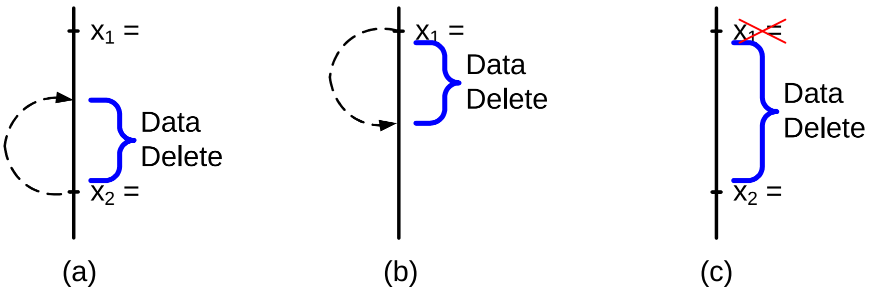


Figure 19: DDelete is used when (a) an instruction is moved earlier overwriting the existing value of variable x, (b) when an instruction is moved later and (c) a definition is deleted

DMove. The DMove descriptor describes the change in storage location of a data value. This descriptor is used when a code (or data) transformation changes the storage location (memory or register) of a data value. Register allocation, performed by a SDT client (e.g., a dynamic optimizer) can result in DMove.

4.3 Example

Figure 20 uses a SPARC code snippet as an example to illustrate transformation descriptors for code that is dynamically optimized. Figure 20(a) shows untranslated code, and Figure 20(b) shows the translated counterpart. During dynamic optimization, several things happen: (1) the `clr` instruction at unoptimized location `0x1bcc` is eliminated; (2) the register `%g1` is reassigned to `%o5` at optimized location `0x100cc`; the register `%o5` is also reassigned to `%o1` (see the last operand of `ld` instruction that has been bolded in both unoptimized and optimized code). and (3) the `ld` instruction is moved from its original location. In the optimized code, the `ld` instruction at `0x100d0` is preceeded by the `inc` and `cmp` instructions where the expected value in register `%o1` (originally `%o5`) will be unavailable. Finally, trampoline code follows the `add` instruction in the optimized code in locations `0x100e4` to `0x100f4`.

Untranslated Code	Dynamically Optimized Code	Trans Prim
`0x1bc8 ld [%o2+408],%o4`	`0x100c8 ld [%o2+408],%o4`	ID
`0x1bcc clr %o3`		CD
`0x1bd0 sll %o3, 2,` **`%g1`**	`0x100cc sll %o3, 2,` **`%o5`**	ID, DM (%g1)
`0x1bd4 ld [%o2+%g1],%o5`	`0x100d0 inc %o3`	ID, DD (%o5)
`0x1bd8 inc %o3`	`0x100d4 cmp %o3, 0xff`	ID, DD (%o5)
`0x1bdc cmp %o3, 0xff`	**`0x100d8 ld [%o2+%o5],%o1`**	CM
`0x1be0 ble 0x1bd0`	`0x100dc ble 0x100f4`	ID
`0x1be4 add %o4,%o5,%o4`	`0x100e0 add %o0,-16,%o0`	ID
...	**`0x100e4 save %sp, 96, %sp`**	CI
...	**`0x100e8 sethi %HI(SDT),%o1`**	CI
	`0x100ec jmp %o1`	CI
	`0x100f0 or %o1,%LO(SDT),%o1`	CI
	`0x100f4 restore`	CI

(a) (b) (c)

Figure 20: Transformation descriptors from dynamic optimization in a SPARC code snippet

SDT Client	Reference System	Transformation Descriptors
Software security	Dynamo-RIO [47]	CInsert, CDelete, CFlush
Dynamic optimization	Dynamo [5]	CInsert, CDelete, CFlush, CMove, DDelete, DMove
Binary translation	Daisy [28]	CInsert, CDelete, CFlush, DDelete, DMove
Dynamic instrumentation	PIN [59]	CInsert, CDelete, CFlush, DMove

Table 2: Transformation descriptors used by different SDT systems

Transformation descriptors for the dynamically optimized code are shown in Figure 20(c). The code descriptor Identity (ID) is generated for all original instructions except the `clr` instruction that was deleted and the `ld` instruction that was moved. CDelete (CD) is generated for the `clr` instruction. DMove (DM) is generated for register `%g1` at location `0x100cc` to indicate that the storage location `%g1` has been reassigned. DDelete (DD) is generated for register `%o1` at locations `0x100d4` and `0x100d8` (`inc` and `cmp` instructions) where the value of `%o5` is unavailable due to code movement. CInsert (CI) is generated for each instruction in the trampoline code.

Transformation descriptors can be used to describe program transformations made by diverse SDT systems. Table 2 shows the transformation descriptors that describe the actions of several SDT systems including program security checkers (Dynamo RIO), a dynamic optimizer (Dynamo), a binary translator (DAISY) and a dynamic instrumenter (Pin).

4.4 Discussion

SDT systems may perform complex transformations that are not directly addressed by the basic transformation descriptors in this chapter. The goal of this section is to give insight into how program trackers should handle new transformations that are not described in this thesis. It should be noted that the purpose of transformation descriptors is not to accurately track program transformations, but rather to hide the effects of transformation from debug users. For any code transformation, the following three guiding principles should be used:

- If multiple computations are combined into one, all the intermediate computations should be invisible to the debug user.
- If a computation is split into multiple computations, the values computed in all but the last computation should be invisible to the debug user.
- If it is difficult to describe a code modification using transformation descriptors or efficient generation of transformation descriptors for a code modification is not possible, err on the side of reportability rather than correctness. That is, if there is a choice between not reporting values versus reporting incorrect values, choose the former.

To illustrate the first two guiding principles, this section describes two complex transformations: splitting of an instruction into multiple instructions and combining of multiple instructions into one. Transformation descriptors are not available to express such transformations. However, the program tracker can generate descriptors as discussed below to describe these transformations.

When two instructions are combined, one way to express this transformation would be to generate two Identity descriptors for each of the original instructions that are associated with the combined instruction. However, expressing the transformation in this way indicates that the code location and data value problems can be solved — breakpoints can be inserted at both the original instructions and all variable values can be reported. This expression (i.e., two Identity descriptors) is incorrect because combining two instructions essentially eliminates the first computation. In other words, the values computed by the first instruction are not computed in the translated code. Therefore, the correct set of descriptors for this transformation is a CDelete for the first instruction and Identity for the second instruction.

Similarly, two Identity descriptors should not be generated when an instruction is split into two. Instead, a CInsert should be generated for the first instruction and an Identity for the second. If the splitted instructions are moved apart (due to code movement), a CInsert should be generated for the first instruction, a CMove for the second instruction and associated DDeletes for the instructions between the first and the second.

4.5 Summary

Program transformation is an inherent part of dynamic translation. Despite apparent functional differences between SDT systems, program transformations made by each SDT system can be dissected into a set of transformation descriptors. This chapter describes the transformation descriptors. This chapter also gives insights into how new transformations should be expressed using transformation descriptors. The next chapter describes how the transformation descriptors can be used to facilitate source level debugging in SDT systems.

Chapter 5. Debug Engine

The debug engine consumes transformation descriptors produced by the program tracker and generates debug information at runtime for use in debugging. Debug information is different from transformation descriptors as the former aids in hiding the effects of program transformations, while the latter describes the transformations. In Tdb, the debug information consists of debug mappings, debug plans and variable values extracted by the debug engine. The debug engine's functionalities include generating and using debug information. Specifically, it has five main functionalities: generating debug mappings, generating debug plans, intercepting native debugger, handling breakpoint and extracting values. Each of these functions is performed by a different component in the debug engine, including the mapping generator, the planner, the execution manager, the breakpoint manager and the runtime information generator. The debug engine stores the debug information and live breakpoint information in a repository called the Debug Information Repository (DIR).

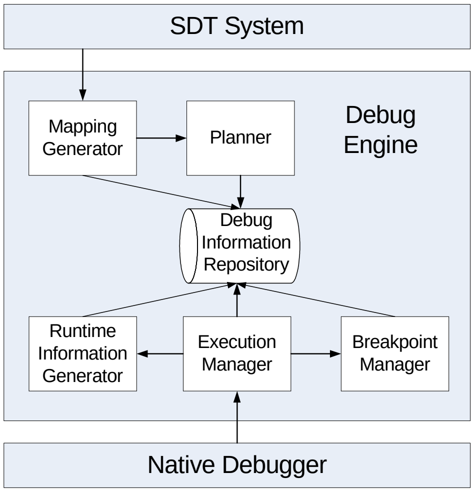


Figure 21: TDB: The Debug Framework

Figure 21 illustrates the organization of the debug engine and its components in the Tdb framework. The mapping generator and the planner components of the debug engine consume the transformation descriptors provided by the SDT system and generate information for use during debugging. The runtime information generator (the RIG), the execution manager and the breakpoint manager components use this information during a debug session to facilitate debugging. The flow of information between the components is facilitated by the debug information repository (DIR) where each component stores information designated for use by another component. The debug engine provides well-defined interfaces for communication with the SDT system and the native debugger

This chapter first describes the interfaces that the debug engine provides to the SDT system and the native debugger. The functionalities of the debug engine components are described next in Section 5.2 – Section 5.7. Section 5.8, shows by means of an example, how the debug engine facilitates source level debugging. Finally, Section 5.9 summarizes the chapter.

5.1 Debug Engine Interfaces

The debug engine provides interfaces that can be targeted by the SDT system to communicate program transformations and accordingly generate debug information. The debug engine also provides interfaces for communication with the native debugger. The debug engine interfaces are described below.

5.1.1 SDT Interface

The SDT system's program tracker communicates transformation descriptors to the debug engine. There are two APIs exposed to the SDT system for communicating code descriptors and data descriptors. The interfaces are shown below.

```
code_descriptor <type, sourceLocation, targetLocation>
```

```
data_descriptor <type, dataValueInfo, locationAfter>
```

For code descriptors Identity and CMove, the `sourceLocation` is the location of the instruction being translated in the program binary (untranslated location), and `targetLocation` is the instruction's location in the code cache. For CInsert, the `sourceLocation` is set to `NULL` and the `targetLocation` is the code cache location of the inserted instruction. For CDelete and CFlush, the `sourceLocation` is the untranslated location of the instruction eliminated/flushed during translation, and the `targetLocation` is set to NULL.

For data descriptors, the `dataValueInfo` consists of a range in which the data value is transformed and the original storage location (register or memory) of the data value. For DMove, the `locationAfter` contains the storage location of the data value after transformation. With DDelete, the data value's liveness is eliminated. Therefore, `locationAfter` is always set to NULL for DDelete.

5.1.2 Native Debugger Interface

The native debugger interface is used for communication of debug commands and queries from the native debugger to the debug engine. In addition, the native debugger interface is used to communicate breakpoints that are hit to the debug engine. In the absence of the debug engine, the native debugger would be invoked when a breakpoint was hit. The APIs in the native debugger interface are shown below. These APIs encompass sufficient detail to allow the debug engine to hide the artifacts of dynamic translation from the native debugger and in turn a debug user.

```
signal_handler <signo>

value := read_value <variable_location>

insert_breakpoint <instruction_location>

remove_breakpoint <instruction_location>

pc := read_pc

write_pc <pc>
```

The debug engine must be informed about any breakpoints (or watchpoints) that are hit in a program instead of the native debugger. The debug engine provides a `signal_handler` interface that is invoked by the native debugger when a breakpoint is hit.

The second interface, `read_value`, provided by the debug engine is used by the native debugger to request a variable's value from the debug engine. In native (untranslated) programs, a debugger would read variable values directly from the program's address space, but Tdb adds a level of indirection via the debug engine. There is no corresponding `write_value` interface. Tdb does not support modification of variable values from within a debug session.

The native debugger also provides interfaces to the debug engine for inserting and removing breakpoints at program locations. The debug engine inserts and removes breakpoints on behalf of the native debugger.

Finally, the debug engine provides interfaces for reading and writing the program counter, i.e., the location at which the program is currently paused. The `read_pc` interface allows the debug engine to report an expected binary location where execution is paused at, instead of a code cache location. The `write_pc` interface enables the debug engine to continue execution in the code cache instead of the location specified by the native debugger. Therefore, reading and writing of the program counter provides the native debugger the complete control of a program's execution. The next section describes how the information communicated to the debug engine is used for facilitating source level debugging.

5.2 Generation of Debug Mappings

Debug mappings are generated by the mapping generator from transformation descriptors. Debug mappings consist of *code location mappings* and *data location mappings*. A code location mapping relates an untranslated or a translated location to another location and helps solve the code location problem. For instance, in response to a breakpoint inserted at an untranslated location, a translated location can be determined from an associated code location mapping. Code location mappings are generated from the code descriptors affecting code locations, including Identity, CInsert, CMove and CDelete descriptors. The data location mappings relate the location of data values in an untranslated application program to a location in the dynamically optimized program. Data location mappings are used when DMove descriptors change the storage location of variables. Mappings are not generated for CFlush and DDelete descriptors because these descriptors lead to removal of code and data values. With CFlush, the mapping generator removes corresponding mappings. The DDelete descriptors require runtime information to reconstruct the "deleted" data values. It is handled by the planner component of the debug engine (see Section 5.3).

`Code Location Mapping`	`<type, headLocation, TailLocations>`
`Data Location Mapping`	`<instructionLocation, locationBefore, locationAfter>`

Table 3: Representation of code location mapping and data location mapping

A code location mapping is a triple shown in the first row of Table 3, consisting of type information (`type`), a location (`headLocation`) and a set of locations (`TailLocations`). A code location mapping can be one of three types: `REGULAR`, `DELETE` and `INSERT`. These types correspond to Identity/ CMove, CDelete and CInsert. The mapping relates the second parameter (`headLocation`) with the third parameter (`TailLocations`). The third parameter is a set to handle code duplication — an untranslated instruction is associated with all duplicate copies of the instruction in the code cache. The code location mappings for the example in Figure 20 on page 43 are shown in Table 6 below. The contents of the table are discussed in conjunction with the mappings later in this section.

A data location mapping associates one storage location with another storage location. Since storage locations (e.g., registers) are often reused for different variable values in different parts of the program, the data location mapping associates storage locations at each instruction location. A data location mapping is a triple, as shown in the second row of Table 4. The first parameter in the triple is the program location where the second parameter (a storage location) is related to the third parameter (another storage location).

Untrans. Binary Location	Binary Location in Code Cache	Trans descriptor	Code Location Mapping: <type, source, Targets>
0x1bc8	0x100c8	Identity	<REGULAR,0x1bc8,{0x100c8}>
0x1bcc		CDelete	<DELETE,0x1bcc,{0x100cc}>
0x1bd0	0x100cc	Identity	<REGULAR,0x1bd0,{0x100cc}>
0x1bd4	0x100d8	CMove	<REGULAR,0x1bd4,{0x100d8}>
0x1bd8	0x100d0	Identity, DMove	<REGULAR,0x1bd8,{0x100d0}>
0x1bdc	0x100d4	Identity, DMove	<REGULAR,0x1bdc,{0x100d4}>
0x1be0	0x100dc	Identity	<REGULAR,0x1be0,{0x100dc}>
0x1be4	0x100e0	Identity	<REGULAR,0x1be4,{0x100e0}>
	0x100e4	CInsert	<INSERT,0x100e4,{0x1bd0}>
	0x100e8	CInsert	<INSERT,0x100e8,{0x1bd0}>

Table 4: Code location mappings for transformation shown in Figure 20 on page 43

The overall functionality of the mapping generator is summarized by the algorithm shown in Table 5. When the mapping generator is invoked with a DMove, it simply invokes the planner for computing debug plans (see Section 5.3). For all other descriptors (code descriptors and the DDelete), the mapping generator computes code location and data location mappings. Table 6 gives the algorithms used by the mapping generator for each of the code descriptors and DDelete. These algorithms are discussed next.

Algorithm

```
∀d // For each descriptor that the mapping generator is invoked with
   if d.type = DMove then
      Planner() // Invoke the Planner: see algorithm in Table 6
   else if d.type = DDelete then
      GenerateMappings <d.type> // Generate mappings
```

Table 5: Summary of mapping generator's actions

5.2.1 REGULAR

For each instruction with Identity or CMove descriptors, a REGULAR code location mapping is generated that associates the untranslated location of the instruction with its translated location. Examples of REGULAR mappings can be seen in Table 4. These correspond to Identity and CMove descriptors.

The algorithm shown in Row 2 of Table 6 is used to generate REGULAR mappings. A REGULAR mapping is generated for all instructions that have an associated Identity or CMove descriptor. If a mapping already exists such that the headLocation of the mapping is the same as the untranslated location of instruction being processed, s, the TailLocations set is appended by adding the translated location of s to it. Otherwise, a new mapping clm is generated with its type set to REGULAR, sourceLocation as s's untranslated location and the translated location of s added to the TailLocations set.

5.2.2 INSERT

For each instruction with a CInsert descriptor, an INSERT code location mapping is generated that associates the instruction with the next instruction in the translated code. The algorithm in Row 3 of Table 6 describes how INSERT mappings are generated. Additionally, each instruction in a trampoline is related to the target of the trampoline. Recall that a trampoline transfers control to the SDT system for translating more code. The target of a trampoline is the untranslated location which is translated next by the SDT system. INSERT mappings are used by the execution manager component of the debug engine to hide the execution of instructions generated during translation and the execution of SDT system itself (see Section 5.4).

The last 2 rows in Table 4 show INSERT mappings for trampoline code. The third parameter in the INSERT mappings is the same address (i.e., 0x1bd0) which is the target of the trampoline.

5.2.3 DELETE

For each instruction with a CDelete descriptor, a DELETE code location mapping is generated. The second row in Table 4 shows a DELETE mapping corresponding to the CDelete descriptor. The algorithm used to generate DELETE mappings is shown in Row 4 of Table 6. The algorithm finds the translated location (loc) of the first "unmoved" instruction that appears later in the instruction stream to the instruction

Transformation descriptor	Algorithm: GenerateMappings <descriptor_type>

Identity / CMove	∀s ∈ ID ∪ CM **//instructions with Identity or CMove**
	  if ∃clm ∈ CLMappings : (clm.headLocation = s.untransLoc) then
	    if (clm.type = REGULAR) then // update existing mapping
	      clm.TailLocations ← clm.TailLocations ∪ {s.cCacheLocation}
	  else
	    clm ← NEW(clmapping) // create new mapping
	    clm.type ← REGULAR
	    clm.headLocation ← s.untransLoc
	    clm.TailLocations ← {s.cCacheLocation}
	    CLMappings ← CLMappings ∪ {clm}

CInsert	clm ← NEW(clmapping) **// create new mapping**
	clm.type ← INSERT
	clm.headLocation ← s.cCacheLocation
	∀s ∈ ID **// instructions with CInsert descriptor**
	  clm.TailLocations ← {s.next.cCacheLocation}
	∀s ∈ exitStubs // instructions in exit stubs
	  clm.TailLocations ← {trampoline.target}
	CLMappings ← CLMappings ∪ {clm}

CDelete	∀s ∈ CD **//instructions with CDelete descriptor**
	  if ∃s′ ∈ Trace : {s′.laterThan(s) ∧ s′ ∩ CM = ∅} then
	    loc ← s′.cCacheLocation // find next code cache instruction
	    if ∃clm ∈ CLMappings : (clm.headLocation = s.untransLoc) ∧
	      if (clm.type = DELETE) then // update existing mapping
	        clm.TailLocations ← clm.TailLocations ∪ {loc}
	    else
	      clm ← NEW(clmapping) // create new mapping
	      clm.type ← DELETE
	      clm.headLocation ← s.untransLoc
	      clm.TailLocations ← {loc}
	      CLMappings ← CLMappings ∪ clm

CFLush	∀clm ∈ CLMappings : s.cCacheLocation ∈ clm.headLocation CLMappings ← CLMappings − clm // update Code location mappings ∀clm ∈ CLMappings : s.cCacheLocation ∈ clm.TailLocations clm.TailLocations ← clm.TailLocations − s.cCacheLocation ∀ dlm ∈ DLMappings : s.cCacheLocation ∈ dlm.instructionLoc DLMappings DLMappings − dlm // update data location mappings Planner(s.cCacheLocation) // invoke debug engine’s planner
DMove	DLMappings ← DM

Table 6: Algorithms to generate code location and data location mappings

with CDelete. The untranslated location of the deleted instruction is then related with `loc`. It is possible for the same instruction to be deleted multiple times (e.g., code duplication followed by deletions). Therefore, the algorithm appends `loc` to `TailLocations`. Section 5.5 describes how `DELETE` is used.

This algorithm for `DELETE` assumes that an “unmoved” instruction is found in the instruction stream following the deleted instruction. If such an instruction is not found, a `DELETE` will not be generated. This scenario will only occur when instructions at the end of a trace are deleted. Therefore, it is expected that such inaccuracies will be minimal.

5.2.4 Deletion of Mappings

When a CFlush descriptor is encountered, the mapping generator removes all the associated code location mappings and data location mappings. In addition, the planner component of the debug engine is invoked so that it can remove the associated debug plans.

5.2.5 Data Location Mappings

Each DMove descriptor contains a triple consisting of the instruction location, location before register allocation and location after register allocation (see Table 3 on page 51). These locations are updated with code cache locations after code is generated in the code cache of the SDT system. A DMove descriptor is essentially a data location mapping and can be used to relate the original storage location of a data value to a new storage location. The original storage location is that in untranslated code and the new storage location is in the dynamically translated code. Therefore, the algorithm in row 5 in Table 6 simply assigns the set of descriptors, `DM`, to the set of data location mappings, `DLMappings`. Each of the descriptors constitute a data location mapping.

The code location and data location mappings can be used by Tdb's debug engine to handle all the transformation descriptors, except DDelete. The DDelete descriptor requires runtime information and is handled by the planner component of the debug engine. The planner is described next.

5.3 Generation of Debug Plans

The planner component of the debug engine guides extraction of runtime data values. While the planner is invoked during dynamic translation, data value extraction is performed during execution by the runtime information generator. The planner's job is to ascertain what values need to be extracted and when. To understand the role of the planner and to appreciate why data value extraction is needed, consider the example in Figure 22. In the figure, an instruction is moved during dynamic optimization. Suppose, the debugger needs to report the value in register %o3 when execution is paused at location 0x100d4 in the dynamically optimized code. Since the moved instruction at location 0x100d0 has overwritten the earlier value of register %o3, the available value is not the expected one. To respond accurately with the expected value of %o3, the runtime value in %o3 must be saved at 0x100d0 before it is overwritten. To guide the extraction of value in %o3, the planner constructs a debug plan as shown below:

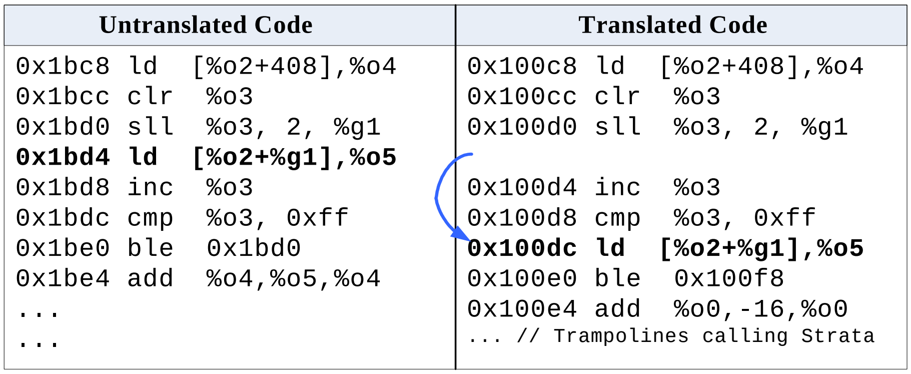
Untranslated Code	Dynamically Optimized Code
0x1bc8 ld [%o2+408],%o4	0x100c8 ld [%o2+408],%o4
0x1bcc clr %o3	0x100cc sll %o3, 2, %o5
0x1bd0 sll %o3, 2, %g1	**0x100d0 inc %o3**
0x1bd4 ld [%o2+%g1],%o5	0x100d4 ld [%o2+%o5],%o1
0x1bd8 inc %o3	
0x1bdc cmp %o3, 0xff	0x100d8 cmp %o3, 0xff
0x1be0 ble 0x1bd0	0x100dc ble 0x100f4
0x1be4 add %o4,%o5,%o4	0x100e0 add %o0,-16,%o0
...	... // Trampoline calling DynOpt
...	... // to translate code at 0x1be8
	0x100f8
	... // Trampoline calling DynOpt
	... // to translate code at 0x1bd0
(a)	(b)

Figure 22: Dynamic optimization moves an instruction to an earlier location

```
Debug Plan: <0x100d0, %o3, {0x100d8}>
```

A debug plan specifies the value that should be recorded at a specific program location; in this case, value in %o3 is recorded just before the instruction at 0x100d0 is executed. The recorded information constitutes a part of the debug information and can be reported until execution reaches `0x100d8`, when the recorded value is discarded.

As illustrated in the example, a debug plan consists of the storage location of a non-reportable variable and a set of instruction locations (`0x100d0` and `0x100d8`) between which the runtime value of variable is not the expected one. Debug plans are generated when DDelete descriptors are encountered. A DDelete implies that code movement has resulted in the actual value of a variable being different than what the user expects.



Untranslated Code

```
0x1bc8 ld   [%o2+408],%o4
0x1bcc clr   %o3
0x1bd0 sll   %o3, 2, %g1
0x1bd4 ld   [%o2+%g1],%o5
0x1bd8 inc   %o3
0x1bdc cmp   %o3, 0xff
0x1be0 ble   0x1bd0
0x1be4 add   %o4,%o5,%o4
...
...
```

Translated Code

```
0x100c8 ld   [%o2+408],%o4
0x100cc clr   %o3
0x100d0 sll   %o3, 2, %g1

0x100d4 inc   %o3
0x100d8 cmp   %o3, 0xff
0x100dc ld   [%o2+%g1],%o5
0x100e0 ble   0x100f8
0x100e4 add   %o0,-16,%o0
... // Trampolines calling Strata
```

(a) (b)

Figure 23: Instruction moved to a later position during dynamic translation

The instruction locations in a debug plan includes a *late point* and a set of *stop* points. A late point is the same location as the original location of the corresponding moved instruction (e.g., `0x100d0` in Figure 22). Stop points are locations where variables defined by the moved instruction are reachable from the late point (e.g., `0x100d8` in Figure 22). When execution reaches a late point, the debug engine rolls ahead (continues) the execution until a stop point is reached. At the stop point, the expected value at the storage location (of a variable) is known. This expected value is reported when queried. The roll ahead is performed by the RIG component of the debug engine without notifying the native debugger. During roll ahead, all values computed in the program and breakpoints encountered are recorded. When execution is continued by the native debugger, the recorded instructions, values and breakpoints are replayed in the order corresponding to the untranslated code.

Consider the example in Figure 23 where an instruction is moved later than its original position. The arrow in the figure shows this code movement. If a breakpoint is inserted at translated location `0x100d8`, the RIG is invoked when execution reaches translated location `0x100d4` and execution is continued and all values computed are recorded until the instruction at 0x100d8 is executed. At this point, the debug engine has enough information to reconstruct expected values and report them to the debug user.

Table 7 gives the algorithm used by the planner. For each instruction with DDelete descriptor (`s’`), a late point is generated as the original position of the corresponding moved instruction `s`. Stop points are locations in the trace where the original definition of `s` is killed. Stop points are generated using the Reachable Definitions analysis developed by Jaramillo [42]. The variable that is defined by `s`, the late point and the set of stop points constitute a debug plan and are added into the DebugPlans set of `s’`. The debug plan is associated with an instruction `s’` only if hasn’t already been associated (to avoid duplicates), denoted by the union operation on DebugPlans set. There can be multiple debug plans at each instruction (for different variables). A simplified algorithm is shown for brevity.

Transformation descriptor	Algorithm

DDelete	`∀(s, s’) ∈ DD` **`// instructions (s’) with DDelete descriptor`**
	`   storageLoc ← s.variable`
	`   latePoint ← s.originalPosition`
	`   StopPoints ← {ReachableDefinitions (s, Trace)}`
	`   debugPlan ← (latePoint, storageLoc, StopPoints)`
	`   DIR.DebugPlans ← DIR.DebugPlans ∪ {debugPlan}`

Table 7: Algorithm used by the Planner to generate debug plans

The notion of late and stop points and the technique of rolling ahead execution when a late point is encountered are borrowed from the Fulldoc debugger [42]. In Fulldoc, the technique of roll-ahead was used in the context of static optimizations.

Discussion. The debug plans are designed for a trace-based SDT system and work only if code movement occurs along a straightline code. In a general control flow graph (CFG), code transformations can move instructions from one path to another. For SDT systems performing transformations on a general CFG, the debug plans will need to be generalized so that code movement across different paths are handled. In particular, data flow analysis can find the paths where code movement renders variable values to be inconsistent and debug plans can be generated on each path. Indeed, such a solution is used in Jaramillo's Fulldoc debugger [42] and Wu's debugger [102].

5.4 Intercepting the Native Debugger

The Execution Manager is the component of Tdb's debug engine that coordinates communication and hides the effects of dynamic optimization from the native debugger. The execution manager is invoked whenever the native debugger performs an action on the program. An action can either be a read/write into program's address space or insertion/removal of a breakpoint/watchpoint. When the native debugger would otherwise write values into the program's address space (or insert/remove breakpoints), the execution manager is invoked to perform the same operations at alternative locations in the code cache. Similarly, when the native debugger reads values from a program's address space, the execution manager is invoked to return alternative values to the native debugger.

The execution manager's invocation mechanism is implementation dependent. As such, the native debugger can be modified to explicitly call the execution manager instead of performing actions on a program. Alternatively, the execution manager can intercept library calls or system calls made by the native debugger. Section 6.1 describes these two implementation strategies in detail. Irrespective of the invocation mechanism, the execution manager is essentially an event-driven system. The execution manager handles three types of events, as summarized in Table 8.

The events handled by the execution manager include (1) a breakpoint hit in the executing program; (2) a request to insert/remove a breakpoint; and (3) a request to read the value of a variable. Each event is accompanied with additional information about the event. For example, when a breakpoint is hit in

Algorithm

```
Input: event, args[] // An event is accompanied with additional data as args

if Event.type = signal then
   signal_handler <args> // Algorithm 1 in Table 8
else if event.type = breakpoint then
   insertRemoveBreakpoint <args> // Algorithm 2 in Table 8
else
   retrieveDataValue <args> // Algorithm 3 in Table 8
```

Table 8: Summary of execution manager’s actions

the program, a signal is raised. The event (signal) is accompanied with the current state of the system (e.g., current program counter location). The execution manager uses this information to take appropriate actions for each event, as shown in Table 9 and discussed below.

5.4.1 Signal Handling

The first algorithm on lines 2-25 in Table 9 shows the actions performed by the execution manager when a breakpoint is hit. The input to the execution manager is the code cache location where a breakpoint was hit. The execution manager uses the DIR and a mode, the ExecutionPhase. The output is an application location which can be provided to the native debugger as the breakpoint location, if requested. In the algorithm, a location value of NULL indicates that execution should be continued without notifying the debugger.

For breakpoints hit in the program, the execution manager consults the DIR to determine whether the breakpoint corresponds to a breakpoint inserted by the native debugger; if so, the native debugger is notified of the breakpoint. A breakpoint may not always correspond to those inserted by the native debugger. Such breakpoints, called *invisible breakpoints*, are inserted at *late* points, *stop* points and at targets of `INSERT` mappings. When invisible breakpoints are hit, the execution manager does not notify the native debugger and instead takes special actions. When a late point is hit, the execution is rolled ahead using the RIG and the ExecutionPhase is set to *RecordPhase*. Execution is subsequently continued (lines 6-9).

If a *stop* point is reached in the record-phase, the RIG decides and notifies the execution manager whether execution should be continued. A value NULL returned by the RIG indicates that execution

```
    //Algorithm 1: Handle a breakpoint that was hit in program
1   Globals: DIR, ExecutionPhase
2   Input: cCacheLocation
3   Output: appLocation

4   if ExecutionPhase = NORMAL_PHASE then // Normal execution phase
5     if ∃d ∈ DIR.DebugPlans: cCacheLocation = d.latePoint then
6       // LATE point : enter record phase
7       ExecutionPhase ← RECORD_PHASE
8       the RIG (cCacheLocation)
9       appLocation ← NULL
10    else // no late point at cCacheLocation
11      if ∃m ∈ DIR.CLMappings : {cCacheLocation ∈ m.TailLocations} then
12        if (appLocation ← m.headLocation) = NULL then insertInvisible()
13      else
14        appLocation ← cCacheLocation
15  else if ExecutionPhase = RECORD_PHASE then // Record phase
16    if ∃d ∈ DIR.DebugPlans : {cCacheLocation ∈ d.StopPoints} then
17      // STOP point in record phase
18      appLocation ← the RIG (cCacheLocation)
19    else
20      the RIG (cCacheLocation)
21      appLocation ← NULL
22  else if ExecutionPhase = REPLAY_PHASE then // Replay phase
23    appLocation = the RIG (cCacheLocation)

24  // Algorithm 2: Insert or remove breakpoints: call Breakpoint Manager
25  Global: DIR
26  Input: action, appLocation

27  DIR.Breakpoints ← BreakpointManager (action, appLocation)

28  //Algorithm 3: Query variable values; given the unoptimized variable loc
29  Input: variableLocation, cCacheLocation
30  Output: variableValue

31  if ∃m ∈ DIR.DLMappings : {variableLocation = m.oldLocation} then
32    variableLocation ← m.newLocation
33  if ExecutionPhase = REPLAY_PHASE then
34    variableValue ← queryRangeRecords (variableLocation, cCacheLocation)
35  else
36    variableValue ← value(variableLocation)
```

Table 9: Algorithms used by Execution Manager

should be continued (lines 16-19). When the RIG returns a non-NULL value, the execution manager single-steps program execution and invokes the RIG after each instruction. In this way, execution is rolled ahead (lines 20-21).

When execution is in *Normal* phase and the breakpoint is not a late point, the execution manager looks up the code location mapping and determines the untranslated location corresponding to the stopped (code cache) location. The native debugger is invoked if the untranslated location is non-NULL. For instructions inserted by the SDT system (and trampolines), the untranslated location is NULL. In this case, invisible breakpoints are inserted at the target of the INSERT mapping (line 12). The native debugger is also invoked when no code location mappings exist for the stopped location (line 14). The lack of code location mapping indicates a breakpoint hit in untranslated code, such as the SDT system itself.

5.4.2 Insert and Remove Breakpoints

To insert and remove a user breakpoint, the native debugger invokes the execution manager as shown in Algorithm 2 in Table 9. The execution manager simply invokes the breakpoint manager (line 27).

5.4.3 Retrieve Data Values

Algorithm 3 in Table 9 shows the retrieval of variable values from the DIR when a variable is queried in replay phase (line 31-36). First, the data location mappings are used to determine if alternative storage locations exist for the expected value. If an alternative location exists then the `variableLocation` is set to this alternative location (lines 31-32). If the execution is in the replay phase, a data structure called "range records" is consulted for the data value. Range records are used by the runtime information generator to store extracted data values. If the execution is not in replay phase, the value stored in the alternative location is returned to the native debugger (lines 33-34).

5.5 Breakpoint Handling

The breakpoint manager is a component of the debug engine that inserts and removes breakpoints (and watchpoints). When the native debugger initiates a breakpoint insertion or removal in the application code, the breakpoint manager is invoked by the execution manager to perform the same action in the code cache. The breakpoint manager uses the algorithm in Table 10. When a breakpoint is to be inserted or removed at an application location, the breakpoint manager looks up the code location mappings to determine the cor-

```
1    //Algorithm to insert/remove a breakpoint
2    Global: DIR
3    Input: action, appLocation

4    $\forall$m $\in$ DIR.CLMappings : {m.headLocation = appLocation $\wedge$
5                           m.type = REGULAR $\vee$ DELETE}
6      if action = INSERTION then
7        DIR.Breakpoints $\leftarrow$ DIR.Breakpoints $\cup$ {m.TailLocations} // insert bp
8      else
9        DIR.Breakpoints $\leftarrow$ DIR.Breakpoints - {m.TailLocations} // remove bp

10     $\forall$d $\in$ DIR.DebugPlans : {d.ccLoc $\in$ m.TailLocations}
11       if action = INSERTION then
12         DIR.Breakpoints $\leftarrow$ DIR.Breakpoints $\cup$ {d.ccLoc} // insert invisible bp
13       else
14         DIR.Breakpoints $\leftarrow$ DIR.Breakpoints - {d.ccLoc} // remove invisible bp
```

Table 10: Algorithm used by Breakpoint Manager

responding code cache locations (line 4). The breakpoint insertion or deletion is then performed at the code cache locations (lines 5-8).

Breakpoints are inserted in code cache for instructions with INSERT as well as DELETE. With DELETE, an instruction that gets eliminated during translation is associated with the next logical instruction (the one appearing later in the instruction stream). Therefore, a breakpoint at an instruction with DELETE is also inserted at the next logical instruction. In a debug session, if breakpoints are inserted at both of these instructions, the debug engine reports two breakpoints hit in the expected order.

When a breakpoint is inserted at a code cache location with a debug plan, additional invisible breakpoints are inserted by the breakpoint manager. Each debug plan contains a late point and one or more stop points. Invisible breakpoints are inserted at each of these locations (late and stop points), as shown in lines 10-13. The breakpoints are stored in the DIR along with the mappings and debug plans.

When the SDT system is active (program execution is paused for dynamic translation), the breakpoint manager removes all breakpoints in the code cache and re-inserts them before execution resumes. In this way, the SDT system does not accidentally read or overwrite a breakpoint instruction (if breakpoints are implemented in software). This is not shown in the algorithm in Table 10.

5.6 Record Replay

Record-replay is a technique to save the program state during execution and to subsequently replay the same execution in a controlled manner. The RIG is a component in the debug engine that uses record-replay to extract variable values whose computation has been moved during code transformation. When a *late* point is hit, the execution manager sets *ExecutionPhase* to *record phase* and invokes the RIG. Table 11 shows the algorithm used by the RIG.

In the record phase, information about the current instruction is saved, including the code cache location of the instruction, values computed by the instruction and the breakpoints encountered. The instruction location is saved in a sequential list, called *RecordedInstructions*, which is maintained in the DIR. This is shown on line 6. In the algorithm, the operator ⊕ adds an element to the end of a list and *popFirst* removes the first element. If there is a breakpoint at the instruction being recorded, the instruction location is added to another list, called the *RecordedBreakpoints,* which is also maintained in the DIR. This is shown in lines 7 and 8. Values that are live at the instruction are saved into a data structure called *range records* (line 9), in the DIR. Range records are similar to live ranges, except that a range record contains a value of a live variable [23]. At a definition of a variable, a new range record is created that contains the variable's value. The live range is extended until the definition is killed. A re-definition of a variable creates a new live range.

Execution is subsequently continued by the execution manager and the RIG is invoked after executing every subsequent instruction. In this way, the RIG records instructions, values computed and the breakpoints encountered for each of the instructions. When the RIG finds that the instruction being recorded has a late point, it updates a set containing all the late points encountered so far, as shown in lines 10-12. The record phase continues and when a stop point is hit, the corresponding late point is removed from the list (lines 13-14). The *replay phase* starts when no more late points are left, as shown in lines 15 and 16.

```
1   //Algorithm to record and replay an instruction
2   Global: DIR, ExecutionPhase
3   Input: cCacheLocation
4   Output: appLocation, variableValue

5   if ExecutionPhase = RECORD_PHASE then // Record phase
6     DIR.RecordedInstructions $\leftarrow$ DIR.RecordedInstructions $\oplus$ cCacheLocation
7     if cCacheLocation $\in$ DIR.Breakpoints then
8       DIR.RecordedBreakpoints $\leftarrow$ DIR.RecordedBreakpoints $\oplus$ cCacheLocation
9     addToRangeRecords (cCacheLocation)

10    if $\exists$d $\in$ DIR.DebugPlans: cCacheLocation $\in$ {d.latePoint $\cup$ d.StopPoints}
11      if cCacheLocation = d.latePoint then // LATE point : extend record phase
12        RecordedLatePoints $\leftarrow$ RecordedLatePoints $\cup$ {cCacheLocation}
13      else                         // STOP point found: start replay phase
14        RecordedLatePoints $\leftarrow$ RecordedLatePoints - d.latePoint
15        if RecordedLatePoints = $\emptyset$ then
16          ExecutionPhase $\leftarrow$ REPLAY_PHASE

17  else if ExecutionPhase = REPLAY_PHASE then // Replay phase
18    bLocation $\leftarrow$ popFirst (RecordedBreakpoints)
19    $\forall$fLocation $\leftarrow$ popFirst (RecordedInstructions) : bLocation $\neq$ fLocation
20      removeFromRangeRecords (cCacheLocation)
21    if $\exists$m $\in$ DIR.CLMappings : {fLocation $\in$ m.TailLocations} then
22      appLocation = m.headLocation
23    else
24      appLocation $\leftarrow$ fLocation
```

Table 11: Algorithm used by the Runtime Information Generator

In the replay phase, the RIG removes recorded instructions from the RecordedInstructions (using *popFirst*) until the next breakpoint location (*bLocation*) in RecordedBreakpoints (lines 18-19). The range records are updated to reflect values available at the current instruction (line 20). Control is subsequently returned to the execution manager and subsequently to the native debugger. The stopped location is indicated as the actual location of the instruction, as shown in lines 21-24.

In this way, the RIG provides an expected order of instructions, breakpoints and update of variable values to the native debugger. The RIG and breakpoint manager are both invoked by the execution manager which communicates with the native debugger.

5.7 Debug Information Repository

The Debug Information Repository (DIR) is where each debug engine component stores information intended for other components. The information stored in the DIR includes mappings, debug plans, values extracted by the RIG and a list of live breakpoints. The mappings and debug plans are generated and updated during dynamic translation and used during execution. Therefore, the DIR must allow querying of the mappings and debug plans with untranslated program binary locations and code cache locations. The former is used for modifying existing mappings and plans, while the latter is used during execution for debugging. Similarly, the DIR must allow for querying breakpoints with untranslated and translated addresses. The former is used during insertion/removal, while the latter is used when a breakpoint is hit.

The DIR is essentially a communication link for the components of the debug engine. The specific details of the DIR are left to implementation.

5.8 Example

To understand how source level debugging can be performed using Tdb, consider the example in Figure 24 in which the untranslated code is transformed by a dynamic optimizer. Suppose, a user wishes to insert a breakpoint at a source code statement with line number `SL1`, single-step to `SL2` and then query the value of variable `var`. SL1 corresponds to binary instructions `0x1bc8` – `0x1bd4`, `SL2` corresponds to instructions `0x1bd8` – `0x1be0` and `var` is stored in register `%o5`. With Tdb, the mapping generator and the planner generate code location mappings and debug plans, as shown in Table 12. The actions of the native debugger corresponding to user commands and queries are intercepted by Tdb's debug engine.

When the native debugger inserts a breakpoint at untranslated location `0x1bc8`, the debug engine's execution manager determines that the corresponding location in the code cache is `0x100c8`. The execution manager uses the code location mapping shown in Row 1 of Table 12. When the user issues a single-step command, which directs the native debugger to resume execution of the program and gives control back to the user when the source location `SL2` is reached. The native debugger resumes execution and pauses after executing every instruction and determining if the stopped binary location corresponds to

Untranslated Code	Dynamically Optimized Code	Trans Prim
`0x1bc8 ld  [%o2+408],%o4`	`0x100c8 ld  [%o2+408],%o4`	ID
`0x1bcc clr  %o3`		CD
`0x1bd0 sll  %o3, 2,` **`%g1`**	`0x100cc sll  %o3, 2,` **`%o5`**	ID, DM (%g1)
`0x1bd4 ld  [%o2+%g1],%o5`	`0x100d0 inc  %o3`	ID, DD (%o5)
`0x1bd8 inc  %o3`	`0x100d4 cmp  %o3, 0xff`	ID, DD (%o5)
`0x1bdc cmp  %o3, 0xff`	**`0x100d8 ld  [%o2+%o5],%o1`**	CM
`0x1be0 ble  0x1bd0`	`0x100dc ble  0x100f4`	ID
`0x1be4 add  %o4,%o5,%o4`	`0x100e0 add  %o0,-16,%o0`	ID
...	**`0x100e4 save %sp, 96, %sp`**	CI
...	**`0x100e8 sethi %HI(SDT),%o1`**	CI
	`0x100ec jmp %o1`	CI
	`0x100f0 or %o1,%LO(SDT),%o1`	CI
	`0x100f4 restore`	CI

(a) (b) (c)

Figure 24: Transformation descriptors for a dynamically optimized code snippet

source location SL2. In the above example, execution resumes at code cache location 0x100c8. When execution reaches 0x100cc, Tdb's execution manager gains control and notifies the debugger that binary location 0x1bd0 has been reached using Row 3 in Table 12. Note that 0x1bd0 is the instruction following 0x100c8. The instruction at location 0x1bcc (Row 2) is never reached as far as the native debugger is concerned. In this way execution continues until execution reaches code cache location 0x100d0 (Row 5).

When execution reaches 0x100d0 with a debug plan, the execution manager invokes the RIG. The RIG determines that the stop point is at location 0x100d8 and continues execution until the stop point is reached and value in %o1 computed at the stop point is saved. Interestingly, the debug plan shows %o5 as the storage location with DDelete. However, the data location mappings are used (shown in Figure 24) to determine that the storage location %o5 is mapped to %o1. Therefore, if the native debugger tries to find the value in %o5, the execution manager returns the value contained in %o1.

Once the value in %o1 has been determined, the execution manager gives control to the native debugger to indicate that binary location 0x1bd8 has been reached. The native debugger gives control back to the user, showing that source location SL2 has been reached. When the user queries the value of source variable var, the native debugger determines the corresponding binary storage location to be %o5. The value in %o1 is subsequently reported to the user.

Source Location	Untrans. Binary Location	Binary Location in F. Cache	Trans descriptor	Code Location Mapping: <type, uLoc, FCLoc>	Debug Plan: <FCLoc, Storage, Late, Stop>
SL1	0x1bc8	0x100c8	ID	<ID,1bc8,100c8>	
	0x1bcc		CD	<CD,1bcc,NULL>	
	0x1bd0	0x100cc	ID	<ID,1bd0,100cc>	
	0x1bd4	0x100d8	CM	<ID,1bd4,100d8>	
SL2	0x1bd8	0x100d0	ID, DM	<ID,1bd8,100d0>	<100d0,%o5,100d0,100d8>
	0x1bdc	0x100d4	ID, DM	<ID,1bdc,100d4>	<100d4,%o5,100d0,100d8>
	0x1be0	0x100dc	ID	<ID,1be0,100dc>	

Table 12: Code location mappings and debug plans for transformation shown in Figure 20 on page 43

5.9 Summary

This chapter presented the TDB's debug engine, which is responsible for generating and using debug information in a SDT environment. The chapter describes the components of the debug engine and illustrates how they address each code transformation descriptor. Finally, an example is used to illustrate how source level debugging is facilitated with Tdb.

Chapter 6. Implementing a Tdb-based Debugger

The previous chapters describe the structure and functionality of the Tdb framework. This chapter describes strategies to implement a source level debugger based on Tdb. This chapter does not describe a specific implementation. Rather, it shares some interesting experiences and implementation choices that were made along the evolution of the techniques developed in this dissertation. The goal of this chapter is to aid the reader (and implementor of a Tdb-based debugger) in deciding the right course of action when encountering the same implementation alternatives.

During the course of this dissertation research, three debuggers were implemented based upon the Tdb framework, as shown in Table 13. Each implementation was targeted to a different SDT system or a different host system (operating system and instruction set architecture) to evaluate the portability of Tdb. For clarity, these debuggers are referred to as Tdb-1, Tdb-2 and Tdb-3. Tdb-1 and Tdb-3 were targeted to a dynamic instrumenter and a dynamic optimizer based on Strata for the SPARC platform. Tdb-2, however, was initially developed for Pin on x86 platform. Tdb-2's debug engine was later reused, with minor configuration changes, for a dynamic instrumenter based on Strata for the SPARC platform. In each implementation, the organization of the Tdb framework stayed the same, i.e., a SDT system, a debug engine and an existing debugger were organized together. However, the underlying communication mechanisms, the interception of native debugger's commands and the program tracker were implemented differently.

This chapter uses the three debuggers to compare and contrast the design decisions made and implementation choices considered while developing each debugger. First, Section 6.1 describes the address space layout of Tdb's debug engine for each implementation, i.e., what parts of the debug engine

Implementation	SDT system	Native Debugger	Instruction set	Communication mechanism	N. Debugger Interception mechanism
Tdb-1	Dynamic instrumenter (FIST)	Gdb	SPARC	Shared memory	Modification to debugger
Tdb-2	Dynamic instrumenter (Pin)	Gdb	Intel-x86	Shared memory	Intercept ptrace library calls
Tdb-3	Dynamic optimizer (Strata-DO)	Gdb	SPARC	/proc based IPC	Modification to debugger

Table 13: Comparison of three source level debuggers based on Tdb framework

are placed in SDT system's address space and what parts are in the native debugger's address space. Section 6.2 details how the program tracker was realized in each implementation. Section 6.3 discusses what is involved in targeting the debug engine interfaces with a focus on the technique used by each implementation for intercepting the native debugger. Section 6.4 describes implementation details of the debug engine's components. This section also describes the interfaces provided by each debug engine component for other components. Finally, Section 6.6 summarizes this chapter.

6.1 Address Space Layout

Modern operating systems provide support for a two-process model of debugging: the program being debugged executes as a process separate from the debugger. This debugging model is used in all three Tdb implementations, where the native debugger (Gdb) executes as one process and the SDT system executes as another process. Figure 25 illustrates the address space layout of Tdb's debug engine with respect to the SDT system and the native debugger, used in all three implementations. As demarcated by the dotted line in the figure, some components of the debug engine are placed in the address space of the SDT system, while others are in the address space of the native debugger.

The mapping generator and planner components of the debug engine are invoked by the SDT system's program tracker when new code is generated or existing code modified/deleted. Therefore, the map-

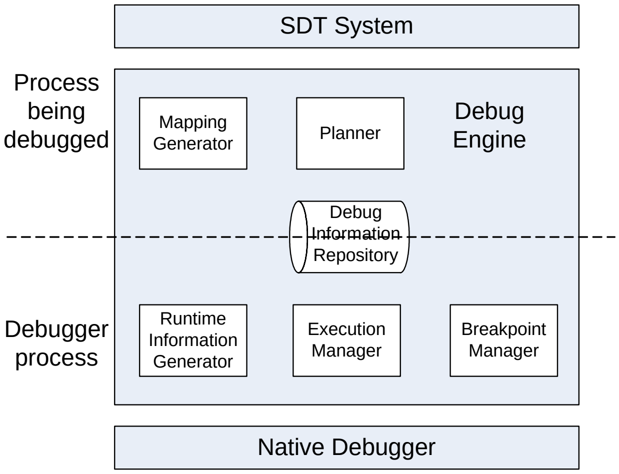


Figure 25: Address space layout of the debug engine

ping generator and the planner are placed in the same address space as the SDT system. In the implementations, these components are compiled into a library and linked against the SDT system.

The execution manager, the breakpoint manager and the runtime information generator components of the debug engine interact with the native debugger. Therefore, these components are placed in the address space of the native debugger.

The DIR is used by all components of the debug engine. Therefore, the DIR is accessible to both processes (the native debugger and the SDT system). Clearly, some form of inter-process communication (IPC) mechanism is needed regardless of which address space holds the DIR. Tdb's efficiency requirement (for generating debug information) motivates the placement of the DIR in the SDT system's address space. The native debugger's actions are based upon an interactive debug user's commands and queries. Therefore, the efficiency requirement for the communication between the DIR and the debug engine's components in the native debugger's address space are not as strict.

The first two implementations, Tdb-1 and Tdb-2, place the DIR in a shared memory segment accessible to both processes. The accessor functions for inserting, querying and removing information in the DIR are also available to both processes. One advantage of using the shared memory mechanism is that the interleaved code generation and execution in SDT systems avoids contention for the shared memory data (debug information). Therefore, accesses to the DIR are quick for both processes.

The disadvantage of the shared memory approach is a practical matter. Shared memory is a scarce resource, and most systems have a predetermined limited size of shared memory segments. Larger segments require intervention of the system administrator. Therefore, scalability and portability of the debug engine suffers when the share memory mechanism is used. The third implementation, Tdb-3, did not use the shared memory mechanism for these reasons.

Tdb-3 places the DIR and all its accessor functions in the address space of the SDT system. This ensures efficient accesses of the DIR from the program tracker. The debug engine components in the native debugger's address space access the DIR using `/proc` based IPC mechanism [60]. Accessing the DIR from the native debugger involves invoking an accessor function in the SDT system's address space using the following four steps.

1. Prior to calling an accessor function, the execution context of the executing program is saved, including the general purpose registers and some machine specific registers such as the condition codes and the program counter;
2. A dummy stackframe is setup before calling the accessor function. The dummy stackframe ensures that if execution stops in the called function (e.g., because of a crash), the native debugger can determine that it was the debug engine's execution and not the program's. The dummy stackframe is set up by extending the program's stack, as shown in Figure 26, so that live data on the stack are unmodified during calls to the accessor functions. Note the unused stack space marked by dotted lines in the Figure 26. The untranslated program binary was generated by the Gcc compiler [89], which occasionally saves live data computed by leaf functions outside the current stackframe. This behavior is not compliant with SPARC ABI. However, to get around intermittent bugs caused by such live data, Tdb-3 left a predetermined amount of stack address space (64 bytes) unused.
3. The accessor functions are called by setting the program counter (PC) to the first instruction in the function and the return address to be NULL; an invalid return address ensures that the debugger is given control after the function has executed.

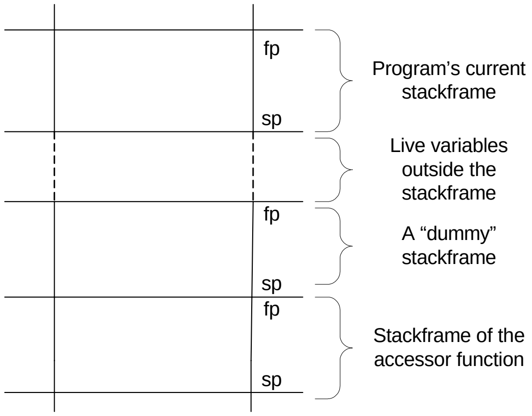

fp
sp
Program's current stackframe
Live variables outside the stackframe
fp
sp
A "dummy" stackframe
fp
sp
Stackframe of the accessor function

Figure 26: Depiction of a program's stack segment, where a dummy stackframe is constructed to make /proc based calls into the program's address space for invoking DIR's accessor functions. A stackframe is defined by its frame pointer (fp) and stack pointer (sp)

4. The returned values are garnered by a `/proc` read operation (read register) and the saved program context is restored.

The main advantage of the `/proc` based approach is that this communication mechanism greatly enhances the scalability of debug information generated and the portability of the debug engine. The implementation effort required in this approach is not significant because Gdb uses the same mechanism on Solaris based systems to allow debug users to call arbitrary functions in the program being debugged. Indeed, future implementations should piggyback the IPC mechanism on the one used by the corresponding native debugger, e.g., `/proc` or `ptrace`.

6.2 The Program Tracker

In general, SDT systems are implemented differently, possibly using different kinds of intermediate representation for performing transformations. The program tracker needs to be specific to each SDT system. Indeed, the Tdb framework does not specify the design of the program tracker for this very reason. Tdb's program tracker can be viewed as a component of a SDT system that uses services provided by other components and install callbacks into other components. This view is illustrated in Figure 27 and used in all three implementations discussed below, i.e., Tdb-1, Tdb-2 and Tdb-3.

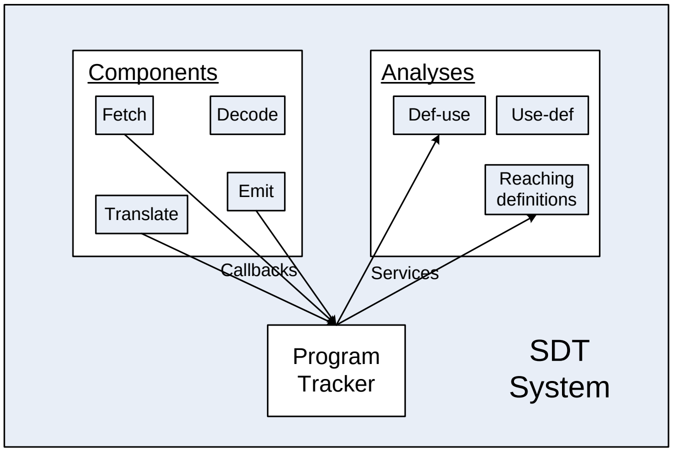


Figure 27: The program tracker installs callback in SDT system's components and uses services provided by the SDT system

6.2.1 Program Tracker for Tdb-1 and Tdb-2

The SDT system used in Tdb-1, FIST, overloads the translate phase of Strata to perform a check to decide whether to instrument the current instruction [49]. The intermediate representation used by FIST is close to the machine instructions and does not carry any semantic information (e.g., the untranslated program binary location for an instruction).

The program tracker, in Tdb-1, installs callbacks in the "fetch" and "emit" stages of the SDT system. When the callback installed on the fetch stage is hit, the program tracker records the untranslated location of instruction and in the emit stage, the corresponding translated location is recorded. Subsequently, an Identity descriptor is generated as `<Identity, untranslated, translated>`.

For each instruction that was not fetched but was emitted, a CInsert is generated as `<CInsert, translated, translated + 1>`. The CInsert results in an `INSERT` mapping that will allow the debug engine to "skip" execution of code at `translated` until `translated+1` is reached. In this way, execution of the instrumented code is kept invisible to a debug user.

The SDT system used in Tdb-2 is another dynamic instrumenter, Pin, on the x86 platform. Pin uses a similar model of instrumentation as FIST. The intermediate representation used by Pin, however, contains the untranslated location for each instruction. The program tracker, in Tdb-2, installs a callback in the "emit" stage of the SDT system. When this callback is hit, Identity and CInsert descriptors are generated based upon whether the untranslated location for an instruction is non-NULL.

6.2.2 Program Tracker for Tdb-3

The SDT system used in Tdb-3 is a dynamic optimizer, Strata-DO [32]. Strata-DO forms instruction traces, constructs a RTL-based intermediate representation, optimizes a trace in several passes and then emits the trace into the code cache. Tracking the effect of all optimization passes on each instruction is too complex to be accomplished simply by installing callbacks at each optimization routine. Traditionally, debuggers for statically optimized code have required significant modification to the optimizer and have resulted in non-linear computation time [8,22,34,41,1,100,102]. With Tdb-3, a new technique of automatically inferring the effect of program transformations is used that determines program transformations by comparing untransformed and transformed instruction traces.

Essentially, the program tracker associates an id, called a statement-id, with each instruction that is fetched and records its untranslated location. The optimizer is modified so that it preserves statement-id's when applying optimizations. For example, if an instruction is moved, the statement-id remains unchanged. If an instruction is duplicated, each of the statements get the same statement-id. After all optimizations have been applied to a trace, the list of statement-id's in the optimized code are compared with the original ones that were recorded during instruction fetch. As a result of this comparison, the code transformations are inferred and the transformation descriptors generated. The algorithm shown in Table 14, `transprim`, describes Tdb-3's program tracker.

```
// 1. Determine all live ranges in trace (a) before optimizations
// are applied; (b) before register allocation is performed; and
// (c) after register allocation if performed. LiveRanges is defined as:
// LiveRanges : {firstInstruction, AllInstructions, storageLocation}
//
// 2. Assign stmt-id to instructions; record their unoptimized locations

// Algorithm 1(a): Compute Original and Actual positions for each stmt
Input: Trace, LiveRangesBeforeOpt, LiveRangesBeforeRA, LiveRangesAfterRA
Output: ID, CI, CD, CM, DD, DM // Transformationdescriptors

∀s : s ∈ Trace ∧ s.moved = FALSE // update moved attribute of insns
  ∀id : (id > s.stmtId) ∧ (id < s.next.stmtId)
    if ∃s’∈ Trace : s’.stmtId = id then
      s’.moved ← TRUE

actualPosition ← 0
∀s : s ∈ Trace // update actual position for all stmts
  actualPosition ← actualPosition + 1
  s.actualPosition ← actualPosition

∀s : s ∈ Trace // update original positions for all stmts
  if (s.moved = TRUE) then
    // find the first instruction on trace with a higher statement-id
    if ∃s’∈ Trace : (s’.stmtId > s.stmtId) ∧ (s’.moved=FALSE) then
      s.originalPosition ← s’.actualPosition
    else
      s.originalPosition ← ∞
  else
    s.originalPosition ← s.actualPosition

// Algorithm 1(b): Compute Identity descriptors
∀s : s ∈ Trace // find all instructions on trace that did not move
  if s.originalPosition = s.actualPosition then
    ID ← ID ∪{s}

// Algorithm 1(c): Compute CInsert descriptors
∀s : s ∈ Trace // find all instructions on trace with stmtId not set
  if s.stmtId = ∅ then
    CI ← CI ∪{s}

// Algorithm 1(d): Compute CDelete descriptors
∀id in [1,lastStmtId] // find all unopt instructions absent in Trace
  ∀s ∈ Trace : s.stmtId ≠ id then
    CD ← CD ∪{(id, untranslatedLocation[id])}

// Algorithm 1(e): Compute CMove descriptors
∀s : s ∈ Trace // find all instructions that moved
  if s.originalPosition ≠ s.actualPosition then
    CM ← CM ∪{s}

// Algorithm 1(f): Compute DDelete descriptors
∀s ∈Trace : s.actualPosition > s.originalPosition
  ∀s’∈{ReachingDefinition (s’, Trace) = s}
    DD ← DD ∪{(s, s’)}

// Algorithm 1(g): Compute DMove descriptors
∀lb : lb ∈ LiveRangesBeforeRA
  if ∃la ∈ LiveRangesAfterRA : (la = lb) then
    ∀s : s ∈ Trace ∩ l.AllInstructions
      DM ← DM ∪ {(s.untransLoc, lb.storageLocation, la.storageLocation)}
```

Table 14: Algorithm to generate Code Transformation descriptors for an optimized trace

Transprim computes transformation descriptors after all optimizations have been applied. Transprim has two pre-processing steps: (1) live ranges of variables are computed before and after register allocation (an optimization used in Strata-DO) and are available for use; (2) each instruction in the unoptimized trace are assigned statement-id's and their untranslated locations are recorded. A statement-id is a unique number associated with instructions and can be assigned in a linear fashion.

Transprim determines an *original* and an *actual* position of each instruction in the optimized trace and uses these positions to compute the transformation descriptors. The original and actual positions of instructions are computed in four steps as shown in Algorithm 1(a). The first step finds the instructions that move due to optimization, as shown in lines 4-7. With code movement, statement-id's are rendered out of order. An instruction is deemed to have moved if its statement-id lies between that of two instructions with higher statement-ids. Other instructions are assumed to have not moved. For example, in a sequence of statement-id's: {1,2,4,5,3,6}, the statement with id 3 is considered to be moved because its statement-id lies between 5 and 6 which are both higher than 3.

Note that this technique does not accurately find which instructions moved. In the above example, it is possible that statement with id 3 was not moved from its position, rather statements with id's 4 and 5 were moved to earlier than 3. In fact, all code movements detected are forward code movements, irrespective of whether the code movement happened in the forward or backward direction. What this technique does, is to find all instructions whose relative positions are different than in the original code. The original order of instructions can be reconstructed by determining the correct (relative) position of instructions considered moved. A debugger can then relate the original positions of instructions with their actual ones. Therefore, detecting moved instructions in the proposed manner suffices for debugging purposes.

In the second step, Algorithm 1(a) linearly assigns actual positions to each instruction in the optimized trace (lines 8-11). The third step of the algorithm determines the original position of each instruction. Intuitively, the original position of an instruction is the position where it would have been, had no code movement taken place. Note that the original position is a position in the optimized code, not the unoptimized code. The algorithm sets the original position for “unmoved” instructions to be the same as their actual positions (lines 12, 19-20). For moved instructions, the algorithm scans the trace to find the first “unmoved” instruction with a higher statement-id. Since the first step marked out-of-order instructions as *moved*, the original position of moved instruction is just before the first “unmoved” instruction with a higher statement-id. The computation of original positions of moved instructions are shown in lines 13-18. Once the original and actual positions have been determined, the code transformation descriptors are computed as described below.

Identity. Any instruction in optimized code that did not move is associated with an Identity descriptor. `Transprim` finds all instructions in an optimized trace whose original and actual positions are the same and adds them to a set of instructions, ID, with Identity descriptor. Algorithm 1(b) in lines 22-24 shows computation of the Identity descriptors.

CInsert. All instructions in the optimized trace that did not have a statement-id assigned to it must be an artifact of dynamic translation. This is because a statement-id is assigned to each instruction in the trace before optimizations. `Transprim` adds instructions without statement-id’s to a set, CI, with CInsert descriptors. The algorithm used for CInsert is shown in Algorithm 1(c) (lines 26-28).

CDelete. All instructions that were deleted during optimization are not present in the optimized trace. `Transprim` checks for statement-id’s in unoptimized trace that are not associated with any optimized instructions, as shown on lines 30 and 31 in Algorithm 1(d). The statement-id not found in the optimized trace along with the corresponding untranslated location are added to a set, CD, with CDelete descriptors (lines 32). Recall that the untranslated location for each statement-id is recorded in the pre-processing stage of `transprim`.

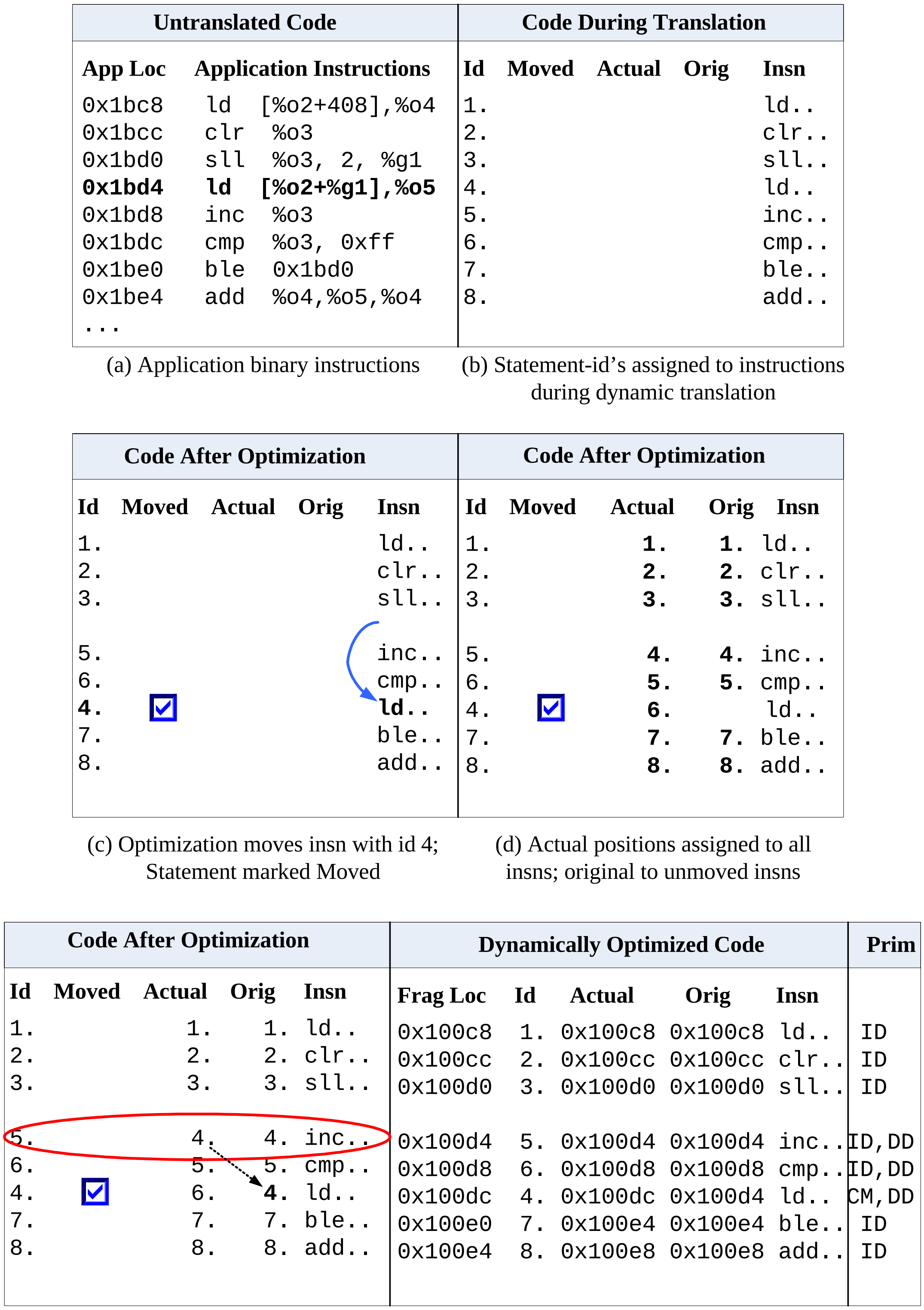
CMove. Any instruction marked as *moved* must have its original position different than its actual position. `Transprim` adds such instructions to a set, CM, with CMove descriptors, as shown in Algorithm 1(e) on lines 34-36. Note that a transformation descriptor describes the summary effect of all transformations applied to an instruction. Therefore, even if an instruction is moved several times during optimization passes, there will be only one CMove descriptor that describes its final position on the trace relative to its original position.

DDelete. A DDelete descriptor is associated with each instruction where a definition is not reachable because the definition (an instruction) was moved. DDelete is generated in conjunction with CMove descriptors. The program tracker performs a reaching definitions analysis for the variable defined by each moved instruction at its original position. Each instruction that is reachable from the original position are associated with a DDelete descriptor. Algorithm 1(f) on lines 38-40 shows the computation of DDelete descriptors, which are then added into a set, DD.

If an instruction is eliminated from a trace (e.g., via dead code elimination) and appropriate compensation code is generated (see Section 2.1.3), the reaching definition analysis assumes that a definition exists at the end of the trace, and analysis is terminated. DDelete descriptors are then associated with all instructions in the trace (except exit stubs with compensation code) following the original position of the moved instruction.

DMove. Register allocation can change storage locations of variables. Algorithm 1(g) determines the new and old storage locations of each live range before and after register allocation is performed. The instruction and its storage locations are added to a set, DM, of DMove descriptors (lines 42-45).

Example. Figure 28 uses an example SPARC code snippet to illustrate how `transprim` generates transformation descriptors. In the example, dynamic optimization of the code snippet leads to exactly one code movement resulting in Identity, CMove and DDelete descriptors. The code snippet is shown in Figure 28(a). The first column of Figure 28(a) shows several application binary locations in the text segment of a program. Column 2 in the figure shows binary instructions at each of the application binary locations. Before optimizations are applied, each instruction in the trace is assigned a unique statement-id. The statement-id’s are shown in Figure 28(b).



Untranslated Code	
App Loc	**Application Instructions**
`0x1bc8`	`ld  [%o2+408],%o4`
`0x1bcc`	`clr  %o3`
`0x1bd0`	`sll  %o3, 2, %g1`
`0x1bd4`	**`ld  [%o2+%g1],%o5`**
`0x1bd8`	`inc  %o3`
`0x1bdc`	`cmp  %o3, 0xff`
`0x1be0`	`ble  0x1bd0`
`0x1be4`	`add  %o4,%o5,%o4`
`...`	

(a) Application binary instructions

Code During Translation				
Id	**Moved**	**Actual**	**Orig**	**Insn**
1.				ld..
2.				clr..
3.				sll..
4.				ld..
5.				inc..
6.				cmp..
7.				ble..
8.				add..

(b) Statement-id's assigned to instructions during dynamic translation

Code After Optimization				
Id	**Moved**	**Actual**	**Orig**	**Insn**
1.				ld..
2.				clr..
3.				sll..
5.				inc..
6.				cmp..
4.	☑			**ld..**
7.				ble..
8.				add..

(c) Optimization moves insn with id 4; Statement marked Moved

Code After Optimization				
Id	**Moved**	**Actual**	**Orig**	**Insn**
1.		**1.**	**1.**	ld..
2.		**2.**	**2.**	clr..
3.		**3.**	**3.**	sll..
5.		**4.**	**4.**	inc..
6.		**5.**	**5.**	cmp..
4.	☑	**6.**		ld..
7.		**7.**	**7.**	ble..
8.		**8.**	**8.**	add..

(d) Actual positions assigned to all insns; original to unmoved insns

Code After Optimization				
Id	**Moved**	**Actual**	**Orig**	**Insn**
1.		1.	1.	ld..
2.		2.	2.	clr..
3.		3.	3.	sll..
5.		4.	4.	inc..
6.		5.	5.	cmp..
4.	☑	6.	**4.**	ld..
7.		7.	7.	ble..
8.		8.	8.	add..

(e) Original position of moved insn is the same as actual position of first insn with a higher statement-id

Dynamically Optimized Code					Prim
Frag Loc	**Id**	**Actual**	**Orig**	**Insn**	
0x100c8	1.	0x100c8	0x100c8	ld..	ID
0x100cc	2.	0x100cc	0x100cc	clr..	ID
0x100d0	3.	0x100d0	0x100d0	sll..	ID
0x100d4	5.	0x100d4	0x100d4	inc..	ID,DD
0x100d8	6.	0x100d8	0x100d8	cmp..	ID,DD
0x100dc	4.	0x100dc	0x100d4	ld..	CM,DD
0x100e0	7.	0x100e4	0x100e4	ble..	ID
0x100e4	8.	0x100e8	0x100e8	add..	ID

(f) After code generation in fragment cache, Original and Actual positions are replaced by fragment cache locations and Transformation primitives computed

Figure 28: Algorithm in Table 14 is used to generate code transformation descriptors for dynamically optimized code. The instruction at untranslated location 0x1bd4 (see (a) above) is moved during optimization. DMove descriptors are not shown in the example above.

The first step in Algorithm 1(a) of `transprim` is to find moved instructions and mark them. In Figure 28(c), the `ld` instruction (originally at location `0x1bd4` in Figure 28(a)) is moved during optimization. The code movement is depicted by the arrow in the figure. Figure 28(c) also shows the statement-id's of the instructions after all optimizations have been applied. Note that the statement-id 4 is out of order and is therefore marked as moved. The second step in Algorithm 1(a) assigns actual positions to each instruction and the third step assigns original positions. The actual positions are marked in a linear sweep through the optimized code. The original position of "unmoved" instructions are set to be the same as their actual positions. The original and actual positions for the optimized code are shown in Figure 28(d). For instructions that move during optimization (see instruction with statement-id 4), the original position is assigned to be the actual position of the first "unmoved" instruction with a higher statement-id. In this case, the instruction with statement-id 5 was not moved during optimization. Therefore the original position of moved instruction is set to 4, as shown by the arrow in Figure 28(d). Note that, if the instruction with statement-id 4 had not moved during optimization, its location would have been before the instruction with statement-id 5.

Once the original and actual positions of all instructions are known, the code transformation descriptors are determined. According to Algorithm 1(b) in Figure 28, the Identity descriptor is associated with all but the moved instruction and the CMove descriptor with the moved instruction. The moved instruction defines register `%o5` and the original definition reaches the instructions with statement-id 5, 6 and 7. What this means is that the expected value in `%o5` at instructions with statement-id's 5, 6 and 7 will not be the actual value. DDelete descriptors are associated with these instructions. The transformation descriptors are shown in the last column in Figure 28(f). Note that the abbreviation ID refers to Identity descriptor, CM to CMove and DD to DDelete descriptor.

When optimized instructions are finally emitted in the code cache during code translation, the original and actual positions of each instruction are replaced by their corresponding code cache locations. The actual position of an instruction is replaced by the location where the instruction is placed in the code cache. Note that all the actual positions in Figure 28(f) are the same as the corresponding code cache location. The original position `p` for each instruction is replaced by the code cache location corresponding to the instruction whose actual position is `p`. Therefore, the original location for any “unmoved” instruction is the same as its actual location. For example, the original location for the first instruction in Figure 28(f) is `0x100c8`, which is the same as its actual location. However, the original location for a moved instruction is the code cache location where the instruction would have been, in the absence of code movement. In Figure 28(f), the original position for instruction at `0x100dc` is `0x100d4`.

Discussion. In Tdb-3, the program tracker generates transformation descriptors with linear complexity. While the algorithm used by the program tracker in Tdb-3 can be used in many other SDT systems, including Tdb-1 and Tdb-2, the complexity of this algorithm is not needed in Tdb-1 and Tdb-2 where transformation descriptors can be determined with significantly lower implementation effort. For example, the program tracker in Tdb-2 was less than 10 lines of code. It did not require modifications to any SDT component (except for a callback) and had linear complexity. The implementation effort in Tdb-1 was similar. The minimal effort in these implementations was due to the presence of only two transformation descriptors, Identity and CInsert, that could be determined relatively easily. In designing the program tracker, the complexity of code transformations should be analyzed and accordingly a solution chosen. Indeed, the algorithm used by Tdb-3’s program tracker will be useful to a large set of SDT systems: those that apply transformations on instruction traces.

6.3 Debug Engine Interfaces

The debug engine provides the SDT interface for communication with the SDT system and the native debugger, as discussed in Section 5.1. The APIs in the SDT interface essentially specify a representation for the code and data descriptors. Consequently, once the program tracker has generated the transformation descriptors, it can easily target these APIs. This section focuses on the debug engine APIs provided by the native debugger interface, because they have interesting implementation choices. The rest of this section discusses how the native debugger interface was targeted in the three Tdb-based implementations.

6.3.1 Targeting the Native Debugger Interface

The APIs provided by the native debugger interface are invoked when the native debugger would otherwise take an action on the program binary (e.g., inserting a breakpoint[1]) or when the program execution pauses due to a breakpoint hit. The implementations discussed in this chapter used one of the following two ways to target the native debugger interface: Tdb-1 and Tdb-3 modified appropriate locations in the native debugger, where an action was performed on the program binary, to explicitly make an API call into the debug engine. Tdb-2 intercepted library calls made by the native debugger and called into the native debugger interface instead. These approaches are discussed below and summarized in Table 15.

Tdb-1 and Tdb3. The native debugger, Gdb, used in Tdb-1 and Tdb-2 is a retargetable and widely used source level debugger [88]. Tdb-1 and Tdb-3 modify Gdb to target the native debugger interface. The advantage of this approach is that it is operating system independent. Retargeting Tdb-1 to another operating system and/or another SDT system requires implementing a new program tracker. The debug engine does not need to be re-implemented as it is already portable.

Targeting the native debugger interface required modification of only seven functions in Gdb, as shown in the second column of Table 15. The native debugger interface is essentially an interface to a library that is linked against the native debugger. The initial implementation effort, that required understanding some internals of Gdb, was less than two weeks.

[1] There are three ways of inserting breakpoints: writing an address into a hardware debug register, overwriting an instruction with a breakpoint trap, and software based breakpoints using jump instructions. In the implementations discussed in this chapter, the first mechanism was used by the native debugger. However, irrespective of which technique is used by a native debugger, they all boil down to writing into a program's address space.

Tdb-2. Tdb-2 was targeted to a commercial environment where portability across debuggers was a requirement. Recompilation of the native debugger was not an option. However, the target platform was fixed: x86/Linux. On Linux, source level debuggers, including Gdb [88] and Idb [40] make use of the *ptrace* [33] system call provided by Linux. Ptrace provides a debugger the ability to read and write into a program's address space and to handle signals on behalf of a program. To read or write into a program's address space, a debugger makes *ptrace* library calls, which in turn call the *ptrace* system call. The debugger also installs signal handlers for `SIGTRAP` (breakpoint/watchpoint), `SIGSEGV` (segmentation fault) and `SIGBUS` (bus error), among others.

Native Debugger Interface	Gdb functions modified in Tdb-1 and Tdb-3	Parameters of intercepted *ptrace* calls in Tdb-2
signal_handler	handle_inferior_event(signal)	// custom signal handler

read_value	target_read_memory(memaddr)
	read_register(regno), regno != PC
		ptrace(GETREGS)
		ptrace(PEEK_DATA)
		ptrace(PEEK_USR)

insert_breakpoint	target_write_memory(memaddr, TRAP)	ptrace(POKE_TEXT, TRAP)
		ptrace(POKE_DATA, TRAP)

remove_breakpoint	target_write_memory(memaddr, !trap)	ptrace(POKE_TEXT, !TRAP)
		ptrace(POKE_DATA, !TRAP)

read_pc	read_pc(memaddr)	ptrace(GETREGS)
write_pc	write_pc(value)	ptrace(SETREGS)

Table 15: Mechanisms used by different implementation to target the native debugger interface. In the table, PC refers to "program counter" and TRAP is a breakpoint instruction.

In Tdb-2, the *ptrace* calls made by the native debugger were intercepted by implementing a dummy function call, also named *ptrace*, that invokes the debug engine instead of the ptrace system call. The dummy *ptrace* was compiled into a shared object that was loaded and dynamically linked against the native debugger. POSIX systems, such as Linux, allow a library to be "preloaded" into a program by using an environment variable `LD_LIBRARY_PRELOAD`. This environment variable was used in Tdb-2 to override original ptrace library calls and instead invoke the "dummy" *ptrace* calls. The third column of Table 15 shows the *ptrace* calls that were intercepted and the corresponding debug engine APIs that were invoked in Tdb-2. In addition to intercepting the *ptrace* calls, Tdb-2 also installed custom signal handlers for `SIGTRAP` to invoke the execution manager. This signal handler overrode the native debugger's signal handler. In this way, any breakpoints that were hit during the execution of the program were reported to the debug engine instead of the native debugger.

6.4 The Debug Engine

The debug engine described in Figure 21 is really the debug engine used in Tdb-3. In Tdb-1 and Tdb-2, the only descriptors generated by the program tracker are Identity and CInsert. Consequently, there is no data value problem. Therefore, the debug engine used in Tdb-1 and Tdb-2 do not have the planner and the RIG components. This section describes some implementation details for the debug engine components and gives the interfaces provided by each component to other components. The interfaces are shown in Table 16.

6.4.1 The Mapping Generator

The mapping generator consumes the transformation descriptors and produces debug mappings. Indeed, the debug engine interfaces exposed to the program tracker, for communicating the transformation descriptors, are really the interfaces of the mapping generator. Note that the interfaces provided by the mapping generator, as shown in the first row of Table 16, are the same as the SDT interfaces described in Section 5.1.

The debug mappings generated by the mapping generator are designed to closely resemble the transformation descriptors (see Section 5.2). This design permits the mapping generator to perform fast processing and quickly generate the debug mappings. The implementation of the mapping generator is straightforward for all transformation descriptors, i.e., the algorithms in Table 6 on page 54 provide the

Debug Engine Component	Component Interfaces
The Mapping Generator	code_descriptor <type, sourceLocation, targetLoc> data_descriptor <type, dataValueInfo, locationAfter>
The Planner	data_descriptor <type, dataValueInfo, NULL>
The Execution Manager	signal_handler <> read_value <variableLocation> insert_breakpoint <instructionLocation> remove_breakpoint <instructionLocation read_pc <> write_pc <>
The Breakpoint Manager	insert_breakpoint <instructionLocation> remove_breakpoint <instructionLocation>
The Runtime Information Generator	perform_recording <codeCacheLocation>
The Debug Information Repository	insertMappingIntoRepos <type, src, targ, isBitmask> insertPlanIntoRepos <{loc}, loc, {loc}, bitmask> insertBreakpointIntoRepos <type, location> recordInstruction <codeCacheLocation> recordBreakpoint <codeCacheLocation> recordLatePoint <codeCacheLocation> insertRangeRecords <variable, value, codeCacheLoc> lookupMappingInRepos <type, codeCacheLocation> lookupPlanInRepos <type, codeCacheLocation> lookupRangeRecords <variableLocation, codeCacheLoc> lookupBreakpoint <type, location> lookupRecordedInstruction <codeCacheLocation lookupRecordedBreakpoint <codeCacheLocation> lookupRecordedLatePoint <codeCacheLocation> removeFromRangeRecords <codeCacheLocation

Table 16: Interfaces provided by the Debug Engine components

sufficient detail. The mapping generator performs one optimization when handling DMove descriptors, which is discussed below.

DMove is typically a result of register allocation, where live ranges are reallocated to a new storage location. Therefore, at any given instruction, there are a number of variables that reside in a different location than the original one. In Tdb-3, the mapping generator generates only one data location mapping per live range. The data location mapping is identical to the DMove descriptor (see Table 6). The mapping generator associates the data location mapping with the first instruction of a live range and associates a bitmap, called *data_avail*, with an appropriate bit set at every other instruction. The bits that are set in *data_avail* represent all variables whose storage location has changed due to program transformation. At runtime, when the execution manager queries the DIR for a storage location, the DIR first checks *data_avail* to decide if the storage location has been relocated. If so, the *data_avail* associated with previous instructions are scanned sequentially until the corresponding data location mapping is found. Note that a sequential traversal of instructions works because the SDT system in Tdb-3 operates at a trace granularity, which is linear in nature.

The mapping generator invokes the planner using the interface shown in row 2 of Table 16 for all DDelete descriptors. The mapping generator stores all the mappings into the DIR using the following interface, where isBitmask determines whether the target is a bitmask (in case of DMove) or a location.

```
insertMappingIntoRepos <type, source, target, isBitmask>
```

6.4.2 The Planner

The planner, only implemented in Tdb-3, handles DDelete descriptors by generating debug plans for each instruction in the live range specified by the descriptor. The algorithm used to generate debug plans, consisting of late and stop points, is described in Table 7 on page 58. For each DDelete, Tdb-3 associates a debug plan with the first instruction in the corresponding live range and a bit is set in *data_avail* for all the other instructions. The technique of setting a bit in *data_avail* is similar to that used by the mapping generator for DMove. The planner's interface to the mapping generator is the same as used by the program tracker to communicate data descriptors, as shown in row 2 of Table 16. The planner uses the following interface to store the debug plans in the DIR, where `ccLoc` is the instruction location where debug plan is attached, `StopPoints` is a set of associated stop points and `bitmask` specifies the variable for which the debug plan is generated. When the `bitmask` is `NULL`, the late and stop points are associated with the `ccLoc` (the late point is `ccLoc`); otherwise, the `bitmask` is associated with the `ccLoc` and stored into the DIR.

```
insertPlanIntoRepos <ccLoc, StopPoints, bitmask>
```

6.4.3 The Execution Manager

The execution manager consists of a `SIGTRAP` handler and the functionality to deal with reads and writes of memory locations and registers. Insertion and removal of a breakpoint is writing a breakpoint trap at a memory location, in all three implementations. The algorithm used by the execution manager is described in Table 9 on page 61. With Tdb-1 and Tdb-2, the signal handler has to only deal with whether a breakpoint was hit at an instruction with `REGULAR` or `INSERT` mapping and take an appropriate action. The signal handler is a stripped down version of the Algorithm 1 in Table 9, consisting of only lines 11-14. In Tdb-3, the entire Algorithm 1, shown in Table 9, comprises the signal handler.

The APIs comprising the native debugger interface are essentially the interfaces of the execution manager, as shown in row 3 of Table 16. The execution manager uses the interfaces provided the breakpoint manager and the RIG (rows 4 and 5 in Table 16). In addition, the execution manager uses the following APIs of the DIR.

```
lookupMappingInRepos <type, codeCacheLocation>

lookupPlanInRepos <type, codeCacheLocation>

lookupRangeRecords <variableLocation, codeCacheLoc>

lookupBreakpoint <type, location>
```

6.4.4 The Breakpoint Manager

The breakpoint manager inserts and removes breakpoints. Internally, the breakpoint manager uses three types of breakpoints: *Original*, *Visible* and *Invisible*. The original breakpoints are breakpoints at program binary locations. When the native debugger tries to insert a breakpoint at a program binary location, its actions lead to the execution manager inserting a breakpoint at the corresponding translated code locations as well as the originally intended (untranslated) locations. Breakpoints are needed at untranslated locations so that if the code containing such a breakpoint gets translated after breakpoint insertion, the breakpoint manager can correctly insert the breakpoint in the translated code.

The visible breakpoints are breakpoints in the code cache that correspond to user breakpoints in the program binary code. The invisible breakpoints are breakpoints inserted by the debug engine at late points, stop points and the target of `INSERT` mappings. All breakpoints are temporarily removed when execution of the translated code pauses for further code generation or for inspection by the native debugger. This ensures that the SDT system and the native debugger do not accidentally read breakpoint instructions instead of the original instructions at program locations containing breakpoints. When execution resumes, all breakpoints are re-inserted.

The breakpoint manager provides interfaces, shown in row 4 of Table 16, to the execution manager and uses the following interfaces of the DIR.

```
lookupMappingInRepos <type, codeCacheLocation>
lookupPlanInRepos <type, codeCacheLocation>
```

6.4.5 The Runtime Information Generator

The runtime information generator, implemented only in Tdb-3, uses the algorithm described in Table 11 on page 65. The data structure used to record values of variables is interesting enough to demand further deliberation.

The runtime information generator uses a modified version of range records data structures that was previously proposed by Coutant [23]. Instead of recording all live data values at each instruction, range records provide an efficient mechanism of recording only those values that change at an instruction. The rest of the values can be reconstructed from earlier information. Coutant’s range records were computed statically to track the storage location of a variable. Tdb-3’s implementation uses the range records for keeping track of data values instead.

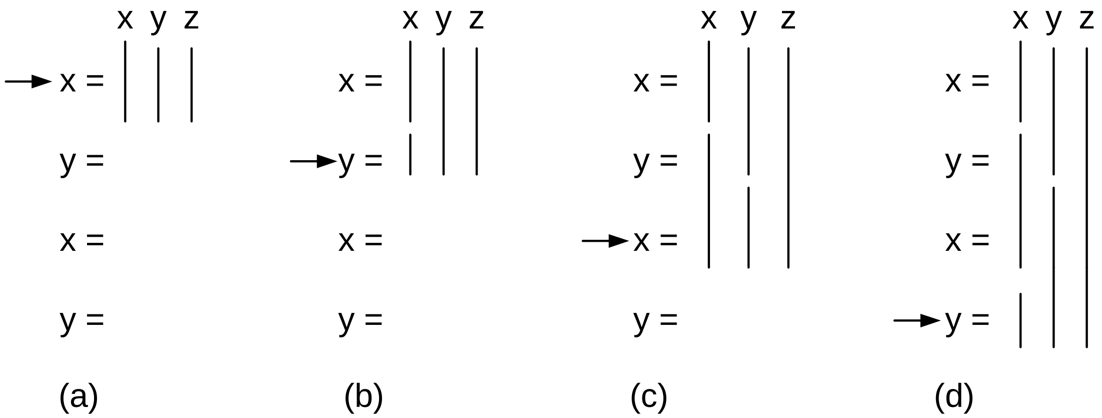


Figure 29: Construction of live ranges for three variables x, y and z as execution progresses through a set of 4 statements

At the start of the record phase, the RIG creates a live range for each variable available at the instruction. Initially, the live range starts and ends at the first recorded instruction. At subsequent instructions in the record phase, if the value of a variable does not change, the live range is extended. If the value of a variable changes, a new live range for the variable is started at that instruction.

The live ranges constructed by the data-value tracker are illustrated by the example in Figure 29. The arrows in the figure show the statement that is about to be executed. The figure shows four instructions, each of which assign a new value to one of the three variables x, y, and z. In Figure 29(a), the execution is at the first instruction. Assume all variables are live at this point. The live ranges for each variable are created and contain the first instruction in the range. When the first instruction executes, the value of variable x changes; therefore a current live range for x ends and a new live range starts, as shown in Figure 29(b). The live ranges for variables y and z are extended. The second instruction defines variable y. If the value of y changes from the existing value when this instruction is executed, a new live range is created for y as shown in Figure 29(c). Similarly, new live ranges are created for variable x and extended for y and z at the third and fourth instructions as shown in Figure 29(d). The live range of variable z encompasses all four instructions, since its value was never changed. During replay, if the value of a variable is queried while replaying an instruction, the range records are consulted to determine the value of the variable.

The runtime information generator provides interfaces, shown in row 5 of Table 16, to the execution manager. The RIG uses the following interfaces of the DIR.

```
recordInstruction <codeCacheLocation>
recordBreakpoint <codeCacheLocation>
recordLatePoint <codeCacheLocation>
insertRangeRecords <variable, value, codeCacheLoc>
lookupMappingInRepos <type, codeCacheLocation>
lookupPlanInRepos <type, codeCacheLocation>
lookupBreakpoint <type, location>
lookupRecordedInstruction <codeCacheLocation
lookupRecordedBreakpoint <codeCacheLocation>
lookupRecordedLatePoint <codeCacheLocation>
removeFromRangeRecords <codeCacheLocation
```

6.4.6 The Debug Information Repository

The DIR is a set of repositories for storing debug information, including debug mappings, debug plans, live breakpoints, range records and lists of instructions and breakpoints recorded by the debug engine's the RIG component. The debug mappings and debug plans are stored in a hashtable structure for quick access. All other information is stored in linked lists. The representation of the debug mappings and debug plans in the DIR merits further discussion.

The DIR views debug mappings and debug plans simply as tuples and is not concerned with their associated semantic information (e.g., the type of code location mapping). The DIR internally maintains two types of mappings: *one-many* and *one-one*. The *one-many* mappings are used to represent `REGULAR` code location mappings, where the target of the corresponding mapping is a set (see Table 6 on page 54). The *one-many* mappings are indexed by untranslated locations, the source of the `REGULAR` mapping, and used during lookups performed by the mapping generator (see Table 6). When a breakpoint is hit and the execution manager queries the DIR for an untranslated location corresponding to a translated location, the DIR provides a reverse lookup of the `REGULAR` mapping. To facilitate this reverse lookup, the DIR generates a *one-one* mapping for each `REGULAR` code location mapping in addition to the *one-many* discussed above. The *one-one* mapping is indexed by the translated location in a `REGULAR` mapping.

The DIR also uses *one-one* mapping for `INSERT` and `DELETE` code location mappings, the data location mappings and the debug plans. The *one-one* mappings for `DELETE` are indexed by untranslated locations, while all other *one-one* mappings are indexed by translated locations. The DIR maintains a separate repository for each type of code location mapping, data location mapping and the debug plans. The memory requirements of the DIR are proportional to the size of code cache used by the SDT system because no more than a code cache full of translated instructions are live at any time.

The DIR provides interfaces to all other components in the debug engine, as shown in Table 16. Each of the DIR interfaces have already been discussed earlier in this section.

6.5 Lessons Learnt

The following discussion summarizes some of the lessons learnt from the three implementations of Tdb.

- The program tracker becomes significantly more complicated when the SDT system performs the code movement transformation. Therefore, if it is known that the SDT system performs little or no code movement, CMoves should not be generated. As discussed in this chapter, the program tracker was less than ten lines of code in Tdb-2, which did not perform code movement.
- The descriptor DDelete can usually be avoided. Any computation that is eliminated by a SDT system is likely not to be queried by a user debugging at the source level. Often deleted instructions are dead code or computation of compiler temporaries, where breakpoints will never hit.

- When retargetability across host systems (operating system, instruction set architecture) is not desired, a better approach to intercepting the native debugger is the one followed in Tdb-2. That is, perform the interception in library code, instead of modifying the source code of the debugger.
- The shared memory model for inter-process communication can be used when the code cache sizes are fixed to a small size (e.g., less than 64 MB). When the code cache size is large or unbounded, this approach is not portable.

6.6 Summary

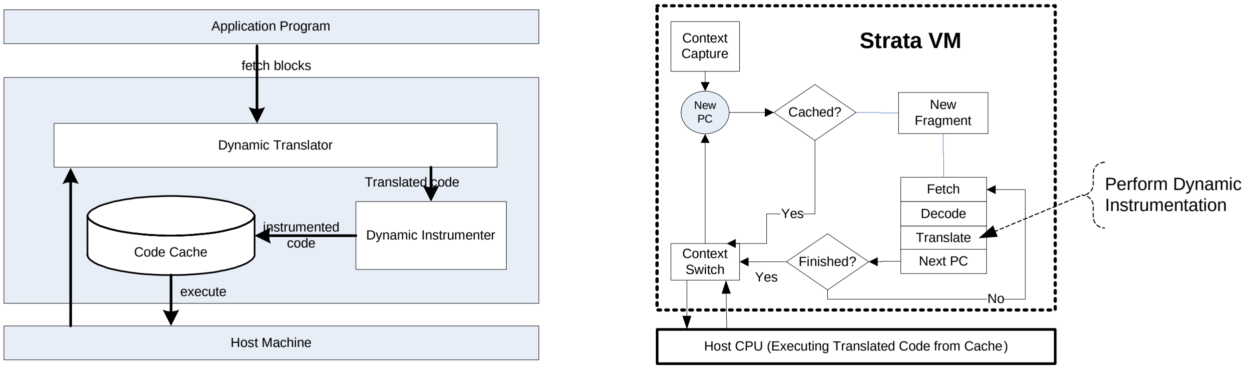
This chapter discussed the experiences and interesting implementation choices that were encountered during the course of this dissertation research. In all, three implementations of the Tdb framework have existed. This chapter compared how the goals and the target SDT system and environment affected some of the implementation decisions. Hopefully, the implementation details discussed in this chapter will help engineers, implementing a Tdb-based debugger, to choose an appropriate solution when presented with the same choices.

Chapter 7. Debugging Dynamically Instrumented Code

Dynamic instrumentation is a powerful technique by which external code can be inserted into an executing program. Dynamic instrumentation has been used for a variety of purposes, including software security [47,80], where the injected code can monitor programs for suspicious behavior, program analysis and profiling [49,59,62,104], where counters are injected at appropriate places in programs, and architectural simulation [19,51], where simulation is performed at certain code locations such as every instruction accessing memory. Dynamic instrumenters can be quickly realized using software dynamic translation. In fact, a number of dynamic instrumenters have been built using SDT including Pin [59], FIST [49] and Dyninst [62].

Despite the usefulness of dynamic instrumenters and relative ease of developing them with SDT technology, techniques for debugging instrumented code at the source level have been lacking. For some instrumented applications, such as software security [47,80] and memory debugging [65], instrumentation code remains in a program while the program is being debugging. With instrumentation code interspersed with regular binary code, it becomes hard for a developer to debug his/her programs. This chapter presents a new debugger, Tdb-1, based upon the Tdb framework to provide debugging support for dynamically instrumented programs. Tdb-1 assumes that dynamic instrumentation is performed in SDT environment.

The first section of this chapter, Section 7.1, briefly describes FIST, a dynamic instrumenter based on Strata, that is used in Tdb-1. The next section, Section 7.2 gives the experimental results. Finally, Section 7.3 summarizes the chapter.



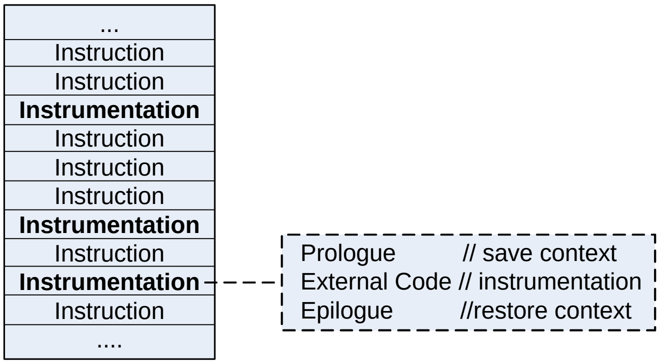
(a) (b)

Figure 30: A dynamic instrumenter can be built by extending a basic SDT system

7.1 FIST: A SDT-based Dynamic Instrumenter

FIST is a dynamic instrumenter that inserts external code into an executing program that is dynamically translated by Strata [49]. Figure 30(a) shows a high level view of FIST, where the dynamic translator communicates with a dynamic instrumenter before code is emitted into the code cache. Figure 30(b) illustrates how the translate stage of a basic SDT system (in this case, Strata) can be overloaded to realize a dynamic instrumenter (in this case, FIST).

FIST inserts external code into at certain locations in translated code. The instrumenter performs a static analysis of code being dynamically translated to determine if it should be instrumented. For instance, a basic block profiler is an instrumentation application in which external code is inserted at the beginning of each basic block [71]. The static analysis in this case would be to determine whether code currently being translated is the target of a branch instruction. When the static analysis determines that instrumentation code should be inserted at a given location, called the *instrumentation point*, instrumentation code is generated along with the usual translated code and emitted into the code cache. In the case of a basic block profiler, the external code increments a counter. In addition to this external code, the instrumentation code



(a) Instruction Stream (b) Code at each instrumentation point

Figure 31: With dynamic instrumentation, additional code is spread across the instruction stream. Each instrumentation point has a prologue, an epilogue and the instrumented external code

also includes a prologue and an epilogue that save and restore the context of execution (registers) so that the external code does not perturb the execution state of the application. Figure 31 illustrates how external code can be interspersed with the original code in the code cache after dynamic instrumentation.

Instrumentation code can be inserted into a program in different ways based upon a number of factors, such as whether code layout changes are permissible or whether instrumentation will need to be removed later. The overheads associated with dynamic instrumentation may also be a concern in some instrumentation systems. FIST uses three different mechanisms to instrument code. These mechanisms are described in detail in Kumar et al. [49]. The following discussion illustrates, by means of an example, two instrumentation mechanisms in FIST that is commonly used by most dynamic instrumenters, including Pin[59], Dyninst [62] and RAIL [12].

7.1.1 Inline versus Fast-breakpoint Instrumentation

On the basis of the position of instrumented external code relative to its instrumentation point, dynamic instrumentation can be classified as *inline* or *fast-breakpoint*. Figure 32 shows the two methods of instrumentation using an example. In the inline method of instrumentation, the external code is placed along with the application code, modifying the original layout of the application code. Figure 32(a) shows the inline method of instrumentation. The fast-breakpoint method of instrumentation, on the contrary, places

Untranslated Code	Dynamically Instrumented Code	Dynamically Instrumented Code
0x1bc8 ld [%o2+408],%o4	**0x100c8 save %sp, -96, %sp**	0x100e4 b,a 0x110a0
0x1bcc clr %o3	**0x100cc sethi %HI(ctr),%o1**	0x100e8 clr %o3
0x1bd0 sll %o3, 2, %g1	**0x100d0 ori %o1,%LO(ctr),%o1**	0x100ec sll %o3, 2, %g1
0x1bd4 ld [%o2+%g1],%o5	**0x100d4 ld %o2, [o1]**	0x100f0 ld [%o2+%g1],%o5
0x1bd8 inc %o3	**0x100d8 add %o2, 1, %o2**	...
0x1bdc cmp %o3, 0xff	**0x100dc sd %o2, [o1]**	**0x110a0 save %sp, -96, %sp**
0x1be0 ble 0x1bd0	**0x100e0 restore**	**0x110a4 sethi %HI(ctr),%o1**
0x1be4 add %o4,%o5,%o4	0x100e4 ld [%o2+408],%o4	**0x110a8 ori %o1,%LO(ctr),%o1**
...	0x100e8 clr %o3	**0x110ac ld %o2, [o1]**
...	0x100ec sll %o3, 2, %g1	**0x110b0 add %o2, 1, %o2**
	0x100f0 ld [%o2+%g1],%o5	**0x110b4 sd %o2, [o1]**
	...	**0x110b8 restore**
		0x110bc ld [%o2+408],%o4
		0x110bc b,a 0x100e8

(a) Original code (b) Inline instrumentation (c) Fast-breakpoint instrumentation

Figure 32: Dynamic instrumentation using inline method and fast-breakpoint method; in both methods, the prologue contains a “save” instruction and epilogue contains a “restore” instruction

external code in a separate location than the original application code. Figure 32(b) shows the fast-breakpoint method of instrumentation. In the fast-breakpoint method, control transfer to the instrumentation code is performed via a branch instruction which replaces the instruction at the instrumentation point. In Figure 32(b), the instrumented code is shown in the box marked as such. The original **ld** instruction is appears to be part of the instrumentation code, which is executed before control transfers back to the application instruction following the instrumentation point. The *payload* function in both methods of instrumentation in Figure 32 refer to an external function invoked as part of instrumentation code.

7.1.2 Removal of Instrumentation

Instrumentation once placed in an application program may eventually need to be removed. For example, a dynamic optimizer may instrument counters into code blocks to detect frequently executed code regions and remove the counter once they reach a threshold. With the inline instrumentation method, removing instrumentation involves either overwriting the instrumentation code with nops or regenerating the code without instrumentation. FIST supports the latter method (regenerating code), because additional nops can increase overheads substantially. Removing the instrumentation code is much easier when fast-breakpoint method is used for instrumentation. The branch instruction that transfers control to the instrumented code can simply be replaced by the original instruction. This method of removal is supported in FIST. Another way to remove fast-breakpoint instrumentation is to regenerate the code without instrumentation, as described before for inline method.

7.2 Experimental Evaluation

Tdb-1 uses FIST as the dynamic instrumenter and Gdb (version 5.3) as the native debugger [88]. FIST uses a fast breakpoint method of instrumentation and the instrumentation, once inserted, is never removed.

Tdb-1 is used in a scenario involving a code security checker. The code security checker can enforce policies on the use of operating system calls, using dynamic instrumentation at system calls[80,79]. For example, the use of file open may be restricted to not open certain files (e.g., the password file). Tdb-1 is validated to ensure that source-level information can be correctly reported. Tdb-1's performance and memory overheads for generating and using the mappings are also evaluated.

7.2.1 Methodology

For the experiments, several SPEC2000 benchmarks were used to compare the results and overhead of *Gdb* and Tdb-1. All experiments were run on a 500 MHz Sun Blade 100 with 256 MB RAM and Solaris 9. Strata's default code cache size of 2 MB was used. The dynamic instrumenter instrumented all system calls to enforce restrictions on operating system services.

Numerous user breakpoints were inserted in the benchmarks with *Gdb* and Tdb-1. To find appropriate breakpoint locations that would likely be hit, the functions that accounted for 90% of the execution time in each benchmark were selected to have breakpoints. Within these hot functions, breakpoint locations were selected at assignment, conditional, and switch instructions. Breakpoints were also inserted at function calls, returns, and instrumented system calls. The number of breakpoints varied from 149–218, with 6–13 functions selected per benchmark.

All benchmarks were run until at least 10,000 breakpoints were hit. The actual number of hits varied depending on the number of system calls that were executed. The breakpoints that were hit covered all dynamic code translations, overhead reduction techniques, and instrumentation points.

7.2.2 Verification

Validating the operation of Tdb-1 required checking that Tdb-1 correctly mapped breakpoint locations to appropriate source instructions. The validation compared the information reported by Gdb without dynamic translation to the information reported by Tdb-1 with dynamic translation. The validation was automatically done by scripts that inserted breakpoints, controlled the program execution, and generated output at each breakpoint. The output from each benchmark run under Gdb and Tdb-1 was also automatically compared.

Table 17 shows the distribution of the breakpoints that were hit for the benchmarks. The table shows the number of unique breakpoints that were hit for the different types of translations. In the table, “Regular” are regularly translated instructions, “Cond” are conditional branches, “Calls” are function calls, “Indirect” are register-indirect branches, and “Instr” is instrumented system calls. For example, in *mcf*, 14 unique breakpoints on assignment instructions were hit a total of 1,569 times. It was verified that *Gdb* and Tdb-1 hit the same breakpoints, in the same order and the same number of times. In all cases, the same breakpoints were hit by both debuggers. It was also verified that the breakpoint commands in both cases reported the same information (e.g., which source line number was hit). The programs were allowed to run to completion when all breakpoints were disabled.

Program	**Number of Unique Breakpoints Hit**					**Number of Breakpoints Hit**				
	Regular	**Cond.**	**Calls**	**Indirect**	**Instr.**	**Regular**	**Cond.**	**Calls**	**Indirect**	**Instr.**
mcf	14	7	15	8	10	1569	2018	3348	3065	382
gcc	24	15	32	7	6	4583	1467	3051	899	2501
gzip	8	3	9	4	9	1804	1219	5404	1572	65
bzip	3	3	6	6	9	1667	1667	3333	3333	76
twolf	32	9	33	14	14	4649	424	3602	1325	566
vortex	3	5	13	5	12	1132	923	5327	2618	1501
vpr	5	6	6	13	27	3174	1005	4898	114	498

Table 17: Number and type of breakpoints hit

Program	**Execution Time (secs.)**		**Total Breakpoints**		**Number of Mappings**			**Memory (kilobytes)**
	GDB	**TDB**	**GDB**	**TDB**	**Identity**	**CInsert^1**	**CInsert^2**	
mcf	183	283	1,941,434	4,280,539	6,081	186	1,701	56
gcc	244	354	2,737,719	6,222,586	174,796	6,806	52,065	1,634
gzip	164	234	1,680,855	3,736,738	8,154	230	1,930	74
bzip	135	192	1,511,400	3,195,215	9,060	220	2,210	82
twolf	191	379	1,996,974	5,382,900	42,580	1,999	8,568	382
vortex	156	329	1,736,651	5,557,198	116,013	683	21,074	1,015
vpr	153	231	1,774,162	3,843,540	29,424	1,548	6,340	267

Table 18: Run-time performance, number of breakpoints, number of mappings, and memory overhead

7.2.3 Performance and Memory

To evaluate performance and memory overhead, the run-times of both debuggers were compared and the memory requirements of Tdb-1 was measured. Table 18 shows the run-times for the benchmarks under *Gdb* and Tdb-1 when breakpoints are inserted and hit, according to the methodology in Section 7.2.1. The first two table columns report run-time in seconds for hitting at least 10,000 user breakpoints. As the table shows, *Gdb* has run-times that range from 135 to 244 seconds and Tdb-1 has run-times from 192 to 379 seconds. Tdb-1 incurs an additional overhead of 42% (*bzip*) to 110% (*vortex*), with an average of 63%, over *Gdb*. This extra overhead is due to generating and using mappings and inserting additional breakpoints in the translated code. To determine where Tdb-1 spends most of the debug time, the overhead due to generating and using the mappings were measured and found to be negligible, accounting for less than 1% of the overhead. The cost of translation and instrumentation was also negligible. The main cost is the insertion of additional breakpoints.

As Table 18 shows in the third and fourth columns (“Total Breakpoints”), Tdb-1 inserts many more breakpoints than *Gdb*. The number of additional breakpoints inserted is increased by 111% (*bzip*) to 220% (*vortex*), with an average increase of 141%. More breakpoints are inserted by Tdb-1 due to code duplication in Strata. Tdb-1 inserts breakpoints in the untranslated in addition to those in the translated code, which further increases the number of breakpoints. As the table demonstrates, the amount of extra overhead incurred by a benchmark directly tracks the number of additional breakpoints inserted.

The number of breakpoints is high for both debuggers due to an implementation artifact. When a breakpoint is hit at run-time, *Gdb* and Tdb-1 remove all active breakpoints and re-insert them when execution resumes. Both implementations can be improved to remove and insert necessary breakpoints on-demand only as determined by the debug commands issued by the user. Furthermore, in an actual usage scenario, very few breakpoints are active at once, and it has been our experience that the overhead due to breakpoint insertion is not perceivable. From these performance results, the run-time overhead incurred by Tdb-1 over *Gdb* is reasonable, given the large number of breakpoints inserted.

The memory overhead of Tdb-1 is shown in the last four columns of Table 18 (“Number of Mappings” and “Memory”). Tdb-1’s memory requirements are related to the size of the code location mappings and number of breakpoints that are active, with the former dominating. Code location mappings are generated for each code descriptor. In *mcf*, the number of Identity descriptors is 6,081, CInsert descriptors due to instrumentation code (shown as CInsert^1) is 1,701, and CInsert descriptors due to trampoline code (shown as CInsert^2) is 186. The maximum size of the mapping table is limited by the size of Strata’s code cache. Entries in the mapping table need 8 bytes for Identity mappings and 4 bytes for the other mappings. In the worst case, there is one entry per instruction in the code cache. For a 2 MB code cache, there are roughly 500,000 instructions, and the mapping table needs 16 MB of memory. In practice, however, the number of entries in the mapping table is considerably less than the maximum number of instructions. For example, *mcf* has a maximum of 7,968 entries at any time and *gcc* has 233,667 entries, as shown in the table. The average number of entries is 23,425 across all benchmarks. The total amount of memory for the mapping table varies from 56 KB to 1.3 MB (average of 501 KB). The breakpoint table's size is minimal as the table holds only active breakpoints. The memory requirement was less than 1KB in all benchmarks.

7.3 Summary

This chapter described a debugger for dynamically instrumented code, Tdb-1, based on the Tdb framework. Tdb-1 enables source level debugging in dynamic instrumenters by interposing a layer (the debug engine) between a native debugger and the application program being dynamically instrumented. The results show that Tdb-1 can be debugged at the source level with minimal performance and memory overheads. The experiments validate the claim that Tdb framework is sufficient to debug dynamically instrumented programs.

Chapter 8. Debugging Dynamically Optimized Programs

Dynamic optimization is the transformation of programs during execution with the goal of improving runtime performance. Dynamic optimizers can be built in hardware as well as software. Among software based dynamic optimizers, JIT based and SDT based optimizers are common. Although, SDT based dynamic optimizers can be relatively quickly realized using a SDT framework such as Strata, source level debugging techniques for such dynamic optimizers currently do not exist. This chapter presents a debugger, called Tdb-3, that uses the Tdb framework to debug dynamically optimized programs.

A SDT based dynamic optimizer typically applies dynamic optimizations on instruction traces [3,5,9,14,64,97]. Traces are particularly well suited for dynamic binary optimization because frequent execution ensures high payoff from optimizations and a trace's straightline control flow simplifies runtime analysis and optimization [3,5,9,14,64,97]. Section 8.1 of this chapter briefly describes a trace based dynamic binary optimizer, Strata-DO, that was used for experimental analysis of Tdb-3. Section 8.2 presents the experimental evaluation of Tdb-3. The experiments show that nearly all values can be reported from a dynamically optimized program with Tdb-3. Section 8.3 summarizes the chapter.

8.1 Strata-DO - A Trace-based Dynamic Optimizer

Strata-DO is a dynamic optimizer built on the Strata infrastructure that optimizes instruction traces at runtime [32]. Due to the changing nature and low cost of trace execution, Strata-DO applies optimizations continuously and frequently as new traces are generated, and existing traces are deleted, combined and/or optimized again (re-optimized). Strata-DO performs several optimizations, including constant propagation, copy propagation, redundant load removal, redundancy elimination, partial redundancy elimination, dead code elimination, partial dead code elimination, partial loop peeling and loop invariant code motion. It also re-optimizes and combines traces during execution. Strata-DO targets the SPARC V9 instruction set.

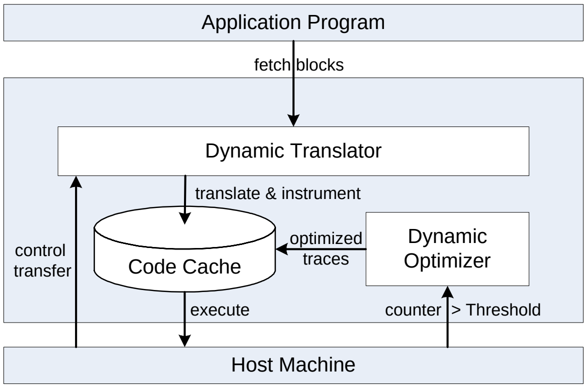


Figure 33: A trace-based dynamic optimizer

Figure 33 shows the structure of Strata-DO. Strata-DO intercepts the execution of a program, optimizes and executes it from a software-managed code cache. The *dynamic translator* component in the figure is a basic SDT system (see Section 2.1.1) that intercepts the executing program to fetch code blocks one at a time, insert counters and emit translated blocks into the code cache from where they execute. The counters keep track of the frequency of execution of the block. After a block of code has executed, the dynamic translator regains control and fetches the next block that executes. When a counter in a code block reaches a threshold, the *dynamic optimizer* is invoked. The dynamic optimizer constructs instruction traces starting at the frequently executed code block. Traces are single-entry and multiple exit entities. A trace exit is a “stub” that transfers control to either the dynamic translator or other traces.

8.2 Implementation and Experimental Evaluation

Tdb-3 was implemented using the implementation discussed in the previous chapter (see Table 13). Tdb-3 integrates Strata-DO [32] and the Gdb debugger [88] with a debug engine. Two sets of experiments were performed to evaluate the effectiveness in reporting values and the efficiency of techniques developed in this research.

The first set of experiments determined how optimizations affect the reportability of values. These experiments assume that all program paths are equally likely. The next set of experiments measured reportability when user breakpoints are hit. These experiments measured "actual" value reportability in a debug session and the performance and memory overheads of Tdb-3. For the experiments, Strata-DO with a default 4 MB code cache was used. A Sun Blade 100 with a 500 MHz UltraSPARC IIe processor with 256 MB of RAM, running Solaris 9, and the reference input sets of the SPECint2000 benchmark suite were used.

To compute the effects of dynamic optimization on the reportability of values, the number of instructions that moved due to optimizations and the variables that were not reportable due to these code movements were counted. The results are shown in Table 19. Column 2 gives the total number of traces that were generated during optimization. Re-optimization in Strata-DO always leads to combining traces. The number of traces varied from 165 to 6333 across the benchmarks. Column 3 shows the percentage of duplicate instructions in the code cache. This number varied from 58% to 69% with an average of 62%. Column 4 shows the number of debug plans generated by the planner, which ranges from 110 to 2439 (average 552). The number of debug plans depend on the number of instructions moved and deleted from paths. Columns 5 and 6 show the percentage of optimized instructions that were moved or deleted. The average percentage of moved and deleted instructions was 2% and 0.5%. Column 7 column shows the increase in reportability due to Tdb-3. The baseline is when the data value problem is not solved. The improvement in reportability ranges from a factor of 2.8 to 34, with an average of 6.4 and a median of 3.

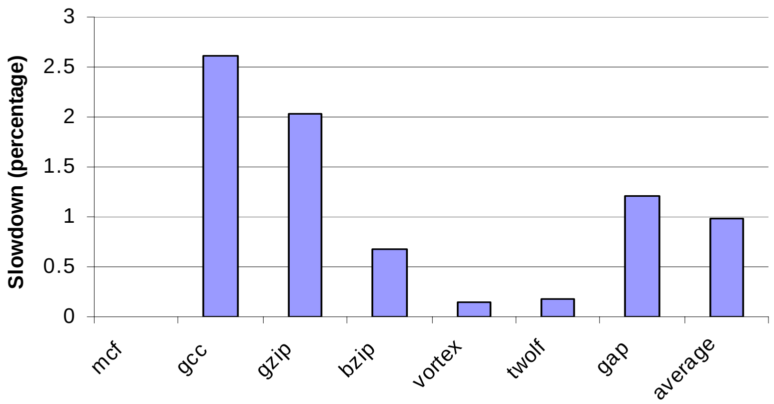


Figure 34: Slowdown due to debug information generation

The percentage of values not reportable in Tdb-3, due to instruction deletion, is shown in the last column and it ranges from 0.01% to 0.1% (average 0.05%). The improvement in reportability is high because optimizations on a trace do not affect reportability at exit stubs.

The overheads of generating Tdb-3's mappings and debug plans were measured next. Programs were run with and without generating debug information and compared the runtimes. Figure 34 shows that the slowdown that ranges from 0% in *mcf* to 2.6% in *gcc* with an average of less than 1%. The overheads are higher for programs that undergo a lot of code translation and code cache flushes. The low overheads of Tdb-3 make it feasible to generate debug information even when the program is not being debugged. This is useful for analyzing core dumps (post-mortem debugging).

The next experiments gathered the debug-time statistics shown in Table 20. For these experiments, breakpoints were inserted at source-level instructions that were moved during dynamic optimization. To get these breakpoint locations, Strata-DO was modified to output the instructions that were moved during a training run, so that the locations from the training run could be used to place breakpoints in an actual run. The inputs to the benchmarks in the training run and the actual run were the same. Out of the potential breakpoint locations, 50 breakpoints per benchmark were randomly selected. Scripts were used to insert breakpoints before execution and continue execution until 10,000 breakpoints hits.

The results from the debug-time experiments are shown in Table 20. Column 2 in the table shows the average number of invisible breakpoints inserted per user-visible breakpoint. These breakpoints were

Table 19: How optimization affects reportability (program paths assumed equally likely)

Benchmark	traces	duplicate	debug plans	moved	deleted	reportability factor	not reportable
mcf	165	64%	134	2.2%	0.6%	30	0.01%
gcc	6333	60%	2439	3%	0.4%	2.8	0.07%
gzip	317	65%	125	1.6%	1%	34	0.03%
bzip	356	69%	241	3%	0.6%	27.9	0.07%
vortex	1232	58%	577	0.7%	0.5%	33	0.1%
twolf	1040	61%	110	2%	0.15%	31	0.03%
gap	1468	58%	239	2.6%	0.004%	32	0.03%

Table 20: Reportability at user breakpoints (runtime)

Benchmark	#invisible	% values not reportable w/o Tdb-3	% values not reportable in Tdb-3	roll-ahead length[a]
mcf	14	67%	8.4%	22
gcc	1.51	5.5%	3.17%	25
gzip	1.38	97%	3.22%	18
bzip	2.3	96%	1.98%	9
vortex	1.9	85%	3.44%	15
twolf	1.6	65%	2.32%	11
gap	1.22	24.5%	3.22%	5

[a] Tdb does not report values which are not computed in translated code, i.e., whose computation is eliminated. However, Tdb never reports an incorrect value of any variable.

inserted due to debug plans and duplicate instructions. The third column shows the percentage of breakpoints hit that had a non-reportable variable due to optimizations, without using Tdb-3. The percentage ranges from 5.5% to 97%, with an average of 62%. Although breakpoints were set at instructions where some variables were not reportable, the numbers in this column are less than 100% because duplicate instructions are often optimized differently. Column 4 shows the percentage of variables at the breakpoints that were not reportable in our framework due to instruction deletion. This ranges from 1.98% to 8.4%, with an average of 3.7%. The percentage of values not reportable is much higher than those in the last column of Table 19 because in our experiments, breakpoints were set at locations where variables are not reportable. Also, in Table 19 the assumption is that all paths are equally likely. Therefore, the percentage of non-reportable values are an upper bound. The last column in Table 20 shows the average length of roll-ahead in every benchmark due to debug plans. The roll-ahead length ranges from 5 to 25 instructions with

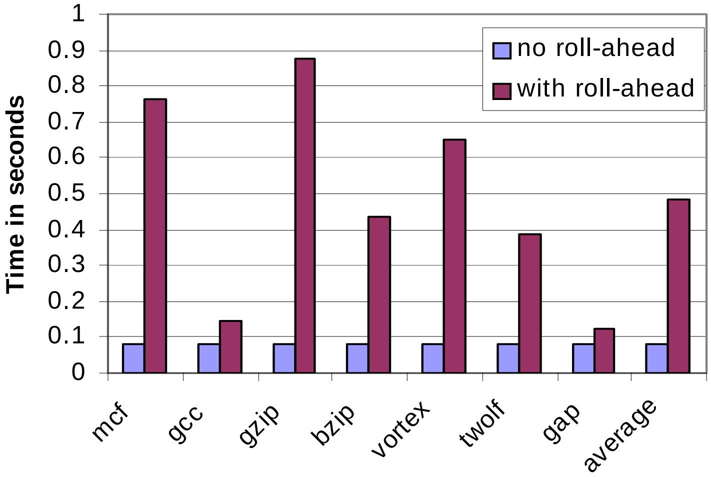


Figure 35: Time to hit one breakpoint without and with roll-ahead

an average of 15 instructions. The results demonstrate that even with breakpoints at instructions that have non-reportable variables, Tdb-3 is able to report a large fraction of variables.

Figure 35 shows the performance overhead of Tdb-3. Figure 35 shows the average time taken by one breakpoint with and without roll-ahead. The time taken to hit one breakpoint when no roll-ahead is needed (i.e., where reportability wasn't affected) was a constant 0.08 seconds. This result is shown using the smaller bars in the figure. When roll-ahead was needed, the record-replay of each rolled-ahead instruction took 0.05 seconds. The average time to hit one breakpoint ranged from 0.12 seconds in *gap* to 0.88 seconds in *gzip*. The difference is due to the difference in roll-ahead frequency and the average roll-ahead length, as shown in columns 3 and 5 of Table 20. Note that in our experimental setup, the breakpoints were inserted specifically at locations where reportability was an issue (leading to roll-ahead). Therefore, these run-times are a rather extreme case. The overheads in Tdb-3 are not noticeable in an interactive debug session.

Figure 36 shows the memory overheads of Tdb-3 using a logarithmic scale. The memory overhead ranges from 69 KB to 2.7 MB, with an average of 685 KB. The memory overheads comprise of debug mappings, debug plans, range records and lists of live breakpoints. The memory consumed changes throughout the execution of a program as new code is generated and older code removed (flushed). The

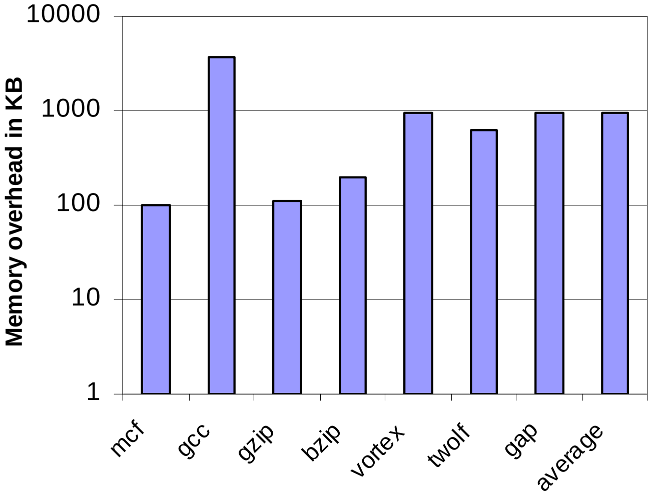


Figure 36: Memory overheads

memory overheads shown in Figure 36 represent the maximum amount of memory consumed during the execution of a benchmark. The code cache size of Strata-DO was set to 4 MB for these experiments, which provides an upper bound to memory overheads. Some benchmarks have a footprint much smaller than the code cache size. For instance, the benchmarks `mcf`, `gzip` and `bzip` led to generation of less than a thousand straightline code blocks, while the benchmark `gcc` filled up the code cache many times during a single execution. Therefore, the memory overhead in the case of gcc was more than an order of magnitude higher than that in the case of `mcf`, `gzip` or `bzip`. The memory overheads of Tdb-3, as a ratio of memory consumed over the amount of code size, are comparable to overheads in debuggers for statically optimized code [8,22,34,41,1,100,102].

8.3 Summary

This chapter described a debugger for dynamically optimized code, Tdb-3, based on the Tdb framework. Tdb-3 enables source level debugging in TDO systems by interposing a layer (the debug engine) between a native debugger and the application program being dynamically optimized. The results show that Tdb-3 can report most program values and has very low overhead. The results also show that Tdb-3's techniques are practical, with a minimal performance overhead of up to just 2.6% when generating debug information during program execution. The experimental results validate that the Tdb framework allows efficient debugging of dynamically optimized programs.

Chapter 9. Conclusions and Future Work

A source level debugger is one of the most important software engineering tools available to a programmer. Irrespective of the programming environment, programmers almost assume the existence of a source level debugger. Currently, source level debugging is largely unavailable to the domain of SDT systems. The acceptance of SDT in mainstream computing, while certainly growing, requires adequate development of software engineering tools. Perhaps more important than any other, a source level debugger would be indispensable.

This thesis is the first work that permits source level debugging targeted to SDT systems. The techniques developed in this research do not inhibit or induce a change in how a SDT system operates when it is being debugged. A framework for debugging, called Tdb, is developed that requires minimal changes to a SDT system and an existing source level debugger (for native programs) to enable debugging of dynamically translated programs. The Tdb framework is highly portable across different SDT systems and host machines. A salient feature of Tdb is that it allows a programmer debugging a dynamically translated program to use his/her favorite source level debugger (for native programs). The programmer, therefore, does not need to learn new debugging commands or a new debugging environment. The techniques developed in this research were implemented in three different SDT systems and experimentally evaluated. The experiments validate that: (1) source level debugging can be efficiently performed in SDT systems; (2) the same debugging techniques can be applied to different SDT systems; and (3) the Tdb framework is highly portable across SDT systems and host machines. This research achieved its goals.

9.1 Summary of Contributions

SDT systems modify programs during execution, thereby rendering static debug information inconsistent with the executing program. If we do not want to restrict the operation of the SDT system, clearly the solution is to generate debug information at runtime as the program is modified and make it available to a debugger for its use. There are three challenges that must be addressed to achieve this solution. First, a systematic approach to determining and representing program modifications is needed. Second, actions that a debugger can use to hide program modifications need to be devised. Finally, a communication mechanism should be developed for informing the debugger about the program modifications. This research addresses all of these challenges to achieve its goals. This thesis makes several research contributions that are described below.

1. A framework, called Tdb, is developed that brings an existing source level debugger together with a SDT system and provides techniques to facilitate source level debugging. The debug engine is highly portable and does not need to be modified when new SDT systems, host machines or debuggers are to be targeted. As long as a SDT system and a debugger interacts with the debug engine through its interfaces, source level debugging can be performed. In this way, the Tdb framework allows different SDT systems and source level debuggers to be plugged together. Tdb provides a *debug engine* that interfaces with the SDT system and the debugger.
2. This thesis describes a notion of *Transformation Descriptors*, that are attributes associated with instructions and data values. A transformation descriptor expresses the combined effect of all transformations that affected the associated instruction or data value. The transformation descriptors are fine-grained — they do not carry any semantic information (e.g., what optimization led to the program transformation). Therefore, the transformation descriptors are portable across SDT systems. In Tdb, a component added to SDT systems, the *program tracker*, dissects the program transformations into transformation descriptors and communicates them to the debug engine.

3. This thesis gives algorithms that are used by the debug engine to generate debug information based upon the transformation descriptors. The debug information is generated whenever new transformation descriptors become available and used in response to a debugging action performed by the debugger. The debug information is used in the following three events. First, if a breakpoint is hit in the translated code, a signal handler in the debug engine handles the breakpoint. Second, if the debugger inserts or removes a breakpoint in the original program, the debug engine uses debug information to insert and remove breakpoints in the translated code. Finally, if the debugger queries the value of a variable, the debug engine determines whether the value is available for reporting and reports the expected value. This thesis gives the algorithms used by the debug engine for all of these three events.
4. This thesis describes three implementations of the Tdb framework and experimentally evaluates two implementations (data for the third one is not available). The experimental evaluation shows that Tdb is effective in debugging dynamically translated programs at the source level and that Tdb's overheads are comparable to the overheads of debuggers of statically generated code. In addition to the experimental evaluation, this thesis also describes and contrasts the experiences in implementing Tdb's techniques in the three debuggers.

9.2 Future Work

There are a number of open and interesting research problems along the lines of this dissertation research. This dissertation research can be evaluated in different systems and the capabilities of the described debug framework can be enhanced. Further, the transformation descriptors developed in this research can be used in applications other than debugging. Several interesting research directions are mentioned below.

1. The transformation descriptors are relatively easy to generate for straightline code sequences (e.g., traces). One area of future work is to target SDT systems that operate on a general CFG. The transfor-

mation descriptors would need to be extended and the mapping generator would have to handle code movement differently. Its not clear whether the overheads of computing transformation descriptors and generating debug mappings would be manageable. One way to reduce the overheads would be to generate debug information on-demand.

2. The transformation descriptors developed in this thesis should be extended to support bytecodes. The descriptors should be combined with the concept of anchor points developed in Wu's research on debugging optimized code to develop generalized transformation descriptors. Generalized transformation descriptors will be able to describe any kind of program transformation and Tdb's debug engine will be able to act as a retargetability layer for native debuggers. If Tdb's techniques can be implemented for a general CFG, it would be possible to use Tdb for JIT compiled programs. Moreover, currently Java and .NET change the behavior of a program when in debug session (e.g., optimizations are disabled); Tdb would enable debugging without the behavioral changes. It would be interesting to debug Java and .NET programs using the same debugger !!
3. A difficult problem in developing SDT systems is that errors in SDT systems often are visible as bugs in the translated program. Tdb's framework can be used to develop software engineering tools that can automatically pin-point errors in the SDT system. One way to approach this problem is by means of comparison checking [42]. An untranslated program is run side-by-side with a dynamically translated program. Breakpoints are set at each source or untranslated binary statement. When a breakpoint is hit, all values computed in the untranslated program are compared with the translated counterpart. As soon as a value is found to be different, a prompt is provided to the SDT system developer for introspection.
4. A SDT infrastructure such as Strata is inherently retargetable. An easy extension of this research would be to retarget the Tdb framework to different instruction set architectures and operating systems.
5. One assumption about SDT systems considered in this thesis was that SDT systems do not modify the semantics of a program (although they can add new semantics). A future area of work would be shed

this assumption. Consider a software updater that loads a newer version of a program module during execution. If the program crashes afterwords, debug support would be appreciated that can unravel whether the program crashed because of the newer software module.

6. Another assumption in this thesis was that the execution paradigm of SDT systems is traditional single-threaded programs. This assumption can be relaxed to target the Tdb framework to a multi-threaded domain or a real time domain.
7. Currently, the debug information generated by Tdb is not maintained in a standard format such as *stabs* [61] or DWARF [94]. An area of future work is to standardize the debug information representation.

Appendix A: References

[1] A. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code", *ACM Conf. on Programming Language Design and Implementation*, 1996.

[2] A. Adl-Tabatabai, "Source-Level Debugging of Globally Optimized Code", *PhD Dissertation*, 1996.

[3] A. Adl-Tabatabai, R. Hudson, M. Serrano and S. Subramoney, "Prefetch injection based on hardware monitoring and object metadata" *ACM Conf. on Programming Language Design and Implementation*, 2004.

[1] B. Alpern, C. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd and S. Smith, "Implementing Jalapeno in Java", *ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications,* 1999.

[2] Apple's website on Rosetta, *http://www.apple.com/rosetta*.

[3] M. Arnold, S. Fink, D. Grove, M. Hind and P. Sweeney, "Adaptive optimization in the Jalapeño JVM", *ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications,* 2000.

[4] M. Arnold and B. Ryder, "A framework for reducing the cost of instrumented code", *ACM Conf. on Programming Language Design and Implementation*, 2001.

[5] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *ACM Conf. on Programming Language Design and Implementation*, 2000.

[6] L. Baraz, T. Devor, O. Etzion, S. Goldernberg, A. Skaletsky, Y. Wang and Y. Zemach, "IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium(R)-based systems", *Int'l Symposium on Microarchitecture*, 2003.

[7] B. Boothe, "Efficient algorithms for bidirectional debugging", *ACM Conf. on Programming Language Design and Implementation*, 2000.

[8] G. Brooks, G. Hansen and S. Simmons, "A new approach to debugging optimized code", *ACM Conf. on Programming Language Design and Implementation*, 1992.

[9] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[10] D. Bruening, V. Kiriansky, T. Garnett and S. Banerji, "Thread-Shared Software Code Caches", *Int'l. Symp. on Code Generation and Optimization*, 2006.

[11] D. Bruening and S. Amarasinghe, "Maintaining consistency and bounding capacity of software code caches", *Int'l. Symp. on Code Generation and Optimization*, 2005.

[12] B. Cabral, P. Marques and L. Silva, "RAIL: Code instrumentation for .NET", *The 20th Annual ACM Symposium on Applied Computing*, 2005.

[13] F. Chang, A. Itzkovitz and V. Karamcheti, “User-level Resource-constrained Sandboxing”, *4th USENIX Windows Systems Symposium Paper*, 2000.

[14] K. Chen, S. Lerner, R. Chaiken and D. Gilles, “Mojo: A dynamic optimization system”, *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[15] A. Chernoff, M. Herdeg, R. Hookway, C. Reev, N. Rubin, T. Tye, S. Yadavalli and J. Yates, “FX!32: A profile-directed binary translator”, *IEEE Micro 18, 2 (Mar.slash April, 1998), 56-64, Presented at Hot Chips IX, 1997.*

[16] T. Chilimbi and M. Hirzel, “Dynamic hot data stream pcomparisonetching for general-purpose programs”, *ACM Conf. on Programming Language Design and Implementation*, 2002.

[17] J. Choi and S. Min, “Race Frontier: Reproducing Data Races in Parallel-Program Debugging”, *ACM Symposium on Principles and Practice of Parallel Programming*, 1991.

[18] C. Cifuentes, B. Lewis and D. Ung, “Walkabout: A retargetable dynamic binary translation framework”, *Workshop on Binary Translation*, 2002.

[19] R. Cmelik and D. Keppel, “Shade: A fast instruction-set simulator for execution profiling”, Technical Report 93–06–06, University of Washington, 1993.

[20] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp, “Dynamic path-based software watermarking”, *ACM Conf. on Programming Language Design and Implementation*, 2004.

[21] J. Cook, “Reverse execution of Java bytecode”, *The Computer Journal*, 2002.

[22] M. Copperman, “Debugging optimized code without being Misled”, *ACM Transactions on Programming Languages and Systems*, 1994.

[23] D. Coutant, S. Meloy, M. Ruscetta, “DOC: a practical approach to source-level debugging of globally optimized code”, *ACM SIGPLAN Conf. on Programming Language design and Implementation*, 1988.

[24] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, “Compiling Java just in time”, *IEEE Micro*, 1997.

[25] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, “The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges”, *Int’l. Symp. on Code Optimization and Generation*, 2003.

[26] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher, “DELI: A new runtime control point”, *Int’l. Symp. on Microarchitecture*, 2002.

[27] D. Dhamdhere and K. Sankaranarayanan, “Dynamic currency determination in optimized programs”, *ACM Transactions on Programming Languages and Systems*, 1998.

[28] K. Ebcioglu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility", *Int'l. Symposium on Computer Architecture*, 1997.

[29] P. Fritzson, P. Aronsson, P Bunus, V. Engelson, L. Saldamli, H, Johansson and A. Karstrom, "The open source modelica project", *The 2th International Modelica Conference*, 2002.

[30] S. Gill, "The diagnosis of mistakes in programmes on the EDSAC", *in Proceedings of the Royal Society, Series A, Volume 206, pp. 538-554*, 1951.

[31] J. Gosling, "Java intermediate bytecodes", *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

[32] A. Guha, J. Hiser, N. Kumar, J. Yang, M. Zhao, S. Zhou, B. Childers, J. Davidson, K. Hazelwood, and M.L. Soffa, "Virtual Execution Environments: Support and Tools", *Workshop on Next Generation Software, International Symposium on Parallel and Distributed Systems*, 2007.

[33] M. Haardt, M. Coleman, *Linux Programmer's Manual*, 1999.

[34] J. Hennessy, "Symbolic debugging of optimized code", *ACM Transactions on Programming Languages and Systems*, 1982.

[35] M. Hicks, J. Moore and S. Nettles, "Dynamic software updating", *ACM Conf. on Programming Language Design and Implementation*, 2001.

[36] J. Hiser, D. Williams, W. Hu, J. Davidson, J. Mars and B. Childers, "Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems" , *ACM Conf. on Code Generation and Optimization*, 2007.

[37] J. Hiser, D. Williams, A. Filipi, J. Davidson, and B. Childers, "Evaluating Fragment Creation Policies for SDT Systems", *ACM Conf. on Virtual Execution Environment*, 2006.

[38] U. Hölzle, C. Chambers and D. Ungar, "Debugging optimized code with dynamic deoptimization", *ACM Conf. on Programming Language Design and Implementation*, 1992.

[39] R. Hood, K. Kennedy and J. Mellor-Crummey, "Parallel program debugging with on-the-fly anomaly detection", *ACM Conf. on Supercomputing*, 1990.

[40] Intel Debugger (IDB) Manual, *Intel Corporation.*

[41] C. Jaramillo, R. Gupta, and M. L. Soffa, "FULLDOC: A full reporting debugger for optimized code", *Proc. of Static Analysis Symposium*, 2000.

[42] C. Jaramillo, "Source level debugging techniques and tools for optimized code", *Ph.D. Thesis, University of Pittsburgh*, 2000.

[43] Java Platform Debugger Architecture, Sun Microsystems, *http://java.sun.com/products/jpda/index.jsp*, 2005.

[44] G. Kc, A. Keromytis, V. Prevelakis, “Countering code-injection attacks with instruction-set randomization”, *Conference on Computer and Communications Security*, 2003.

[45] A. Kennedy and Don Syme, “Design and implementation of generics for the NET common language runtime”, *ACM Conf. on Programming Language Design and Implementation*, 2001.

[46] P. Kessler, “Fast breakpoints: Design and implementation”, *ACM Conf. on Programming Language Design and Implementation*, 1990.

[47] V. Kiriansky, D. Bruening and S. Amarasinghe, “Secure execution via program shepherding”, *USENIX Security Symposium*, 2002.

[48] N. Kumar, B. Childers, D. Williams, J. Davidson and M.L. Soffa, “Compile-time planning for overhead reduction in software dynamic translation”, *Int’l. Jounral on Parallel Programming*, 2005.

[49] N. Kumar, J. Misurda, B. Childers and M.L. Soffa, “Instrumentation in Software Dynamic Translators for Self-Managed Systems”, *ACM Workshop on Self-Managing Systems (WOSS'04) during the ACM SIGSOFT Int’l. Symp. on Foundations of Software Engineering* , 2004.

[50] N. Kumar, B. Childers, M.L. Soffa, “Source-level debugging for dynamically translated programs”, *Technical Report, Department of Computer Science, University of Pittsburgh, TR-05-120, http://www.cs.pitt.edu/~naveen/papers/tdb.pdf*, 2005.

[51] N. Kumar, B. Childers, M.L. Soffa, “Low overhead program monitoring and profiling”, *Technical Report TR-04-156, Department of Computer Science, University of Pittsburgh, http://www.csw.pitt.edu/~naveen/papers/INSOP.pdf*, 2004.

[52] N. Kumar and R. Peri, “Transparent Debugging of Dynamically Instrumented Programs”, *In proceedings of Workshop on Binary Instrumentation and Applications, Held in conjunction with the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[53] R. Lencevicius, U. Hölzle and A. Singh, “Query-based debugging of object-oriented programs”, *ACM SIGPLAN conf. on Object Oriented Programming Systems Languages and Applications*, 1997.

[54] S. Liang and Gilad Bracha, “Dynamic class loading in the Java virtual machine”, *ACM Conf. on Programming Language Design and Implementation*, 1998.

[55] T. Lindholm and F. Yellin, “The Java virtual machine specification”, *Addison-Wesley*, 1996.

[56] M. Linton, “The evolution of dbx”, *Proceedings of the Summer USENIX Conference*, 1990.

[57] J. Lu, H. Chen, P. Yew, W. Hsu, “Design and implementation of a lightweight dynamic optimization system”, *Journal of Instruction Level Parallelism*, 2004.

[58] P. Nelson, “A comparison of PASCAL intermediate languages”, *ACM SIGPLAN Notices 14(8):208213*, 1979.

[59] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation”, *ACM Conf. on Programming Language Design and Implementation*, 2005.

[60] M. McKusick, K. Bostic, M. Karels and J. Quarterman, “The Design and Implementation of the 4.4 BSD Operating System”, Book, 1996.

[61] J. Menapace, J. Kingdon and D. MacKenzie, “The stabs debug format”, *Cygnus Support*, 1993.

[62] B. Miller, “The Paradyn performance tools”, *IEEE Computer*, 1995.

[63] J. Misurda, J. Clause, J. Reed, B. Childers, M. L. Soffa, “Demand-driven structural testing with dynamic instrumentation”, *ACM SIGSOFT Int'l. Conf. on Software Engineering*, 2005.

[64] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal and W. Hwu, “An architectural framework for tuntime optimization”, *IEEE Transactions on Computers 50, 6 (2001), 567-589.*

[65] N. Nethercote and J. Seward, “Valgrind A Program Supervision Framework”, *Electronic Notes in Theoretical Computer Science*, 2003.

[66] R. Netzer and J. Xu, “Adaptive message logging for incremental replay of message passing programs”, *Int’l Conf. on High Performance Networking and Computing held with ACM Conf. on Supercomputing*, 1993.

[67] R. Netzer and B. Miller, “Optimal tracing and replay for debugging message-passing parallel programs”, *The Journal of Supercomputing*, 1995.

[68] J. Norton, “Dynamic class loading in C++”, *Linux Journal*, 2000.

[69] W. Pauw, R. Helm, D. Kimelman and J. Vlissides, “Visualizing the behavior of object-oriented systems”, *ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications*, 1993.

[70] M. Pellegrino, “Improve your understanding of .NET internals by building a debugger for managed code”, *MSDN Magazene, http://msdn.microsoft.com/msdnmag/issues/02/11/CLRDebugging/,* 2002.

[71] K. Pettis, R. Hansen, “Profile guided code positioning”, *ACM SIGPLAN Notices*, 1990.

[72] L. Pollock and M.L. Soffa, “High-level debugging with the aid of an incremental optimizer”, *ACM Workshop on Parallel and Distributed Debugging, 26(4):103-114*, 1991.

[73] M. Prvulovic and J. Torrellas, “ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes”, *Int’l. Symp. on Computer Architecture*, June 2003.

[74] N. Ramsey and D. Hanson, “A retargetable debugger”, *ACM Conf. on Programming Language Design and Implementation*, 1992.

[75] M. Ronsse and K. Bosschere - ACM Transactions on Computer Systems, “RecPlay: a fully integrated practical record/replay system”, *ACM Transactions on Computer Systems*, 1999.

[76] M. Rosenblum, S. Herrod, E. Witchel and A. Gupta, “Computer simulation: The SimOS approach”, *IEEE Parallel and Distributed Technology*, 1995.

[77] M. Rosenblum and T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends”, *IEEE Transactions on Computers*, 2005.

[78] A. Sah, “An efficient implementation of the Tcl language”, *Master’s Thesis, Univ. of Cal. at Berkeley, Tech report UCB-CSD-94-812*, May 1994.

[79] K. Scott, N. Kumar, S. Veluswamy, B. Childers, J. Davidson, M. L. Soffa, “Reconfigurable and retargetable software dynamic translation”, *Int’l. Symp. on Code Generation and Optimization*, 2003.

[80] K. Scott and J. Davidson, “Safe virtual execution using software dynamic translation”, *Annual Computer Security Applications Conference*, 2002.

[81] K. Scott, N. Kumar, B. Childers, J. Davidson, and M. L. Soffa, “Overhead reduction techniques for software dynamic translation”, *invited paper, NSF Workshop on Next Generation Software, during the Int’l. Parallel and Distributed Processing Symposium*, 2004.

[82] E. Schnarr and J. Larus, “Fast out-of-order processor simulation using memoization”, *Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[83] S. Shogan and B. Childers, “Compact binaries with code compression in a software dynamic translator”, *Design Automation and Test in Europe Conference*, 2004.

[84] V. Sreedhar, M. Burke and J. Choi, “A framework for interprocedural optimization in the presence of dynamic class loading”, *ACM Conf. on Programming Language Design and Implementation*, 2000.

[85] A. Srivastava and A. Edwards, “Vulcan: Binary transformation in a distributed environment”, Microsoft Research, MSR–TR–2001–50, April 2001.

[86] J. Sugerman, G. Venkitachalam, B. Lim, “Virtualizing I/O devices on VMWare workstation’s hosted virtual machine monitor”, *USENIX Annual Technical Conference*, 2001.

[87] Y. Sugiyama, “Modifying objects in running java programs”, *IASTED Int’l Conf. on Software Engineering and Applications*, 2001.

[88] R. Stallman and R. Pesch, “Using GDB: A guide to the GNU source-level debugger”, GDB v4.0, Free Software Foundation, Cambridge, MA, 1991.

[89] R. Stallman, “GNU Compiler Collection Internals”, *Free Software Foundation, Reference Manual*, 2002.

[90] The Java HotSpot Performance Engine Architecture, *White Paper, http://java.sun.com/products/hotspot/whitepaper.html*, 1999.

[91] C. Tice and S. Graham. “Optview: A new approach for examining optimized code”, *Workshop on Program Analysis For Software Tools and Engineering*, 2000.

[92] M. Tikir, G. Lueh and J, Hollingsworth, “Recompilation for debugging support in a JIT compiler”, *ACM Workshop on Program Analysis for Software Tools and Engineering*, 2002.

[93] M. Tikir and J. Hollingsworth, “Efficient Instrumentation for Code Coverage Testing”, *International Symposium on Software Testing and Analysis,* 2002.

[94] TIS Committee. DWARF debugging information format specification version 2.0, May 1995.

[95] J. Trojer and J. Gough, “Fast dynamic translation - the yirr-ma framework”, *Workshop on Binary Translation,* 2002.

[96] D. Ung and C. Cifuentes, “Machine adaptable dynamic binary translation”, *ACM Workshop on Dynamic Optimization*, 2000.

[97] M. Voss and R. Eigenmann, “A framework for remote dynamic program optimization”, *ACM Workshop on Dynamic Optimization,* 2000.

[98] J. Whaley, “Joeq: A virtual machine and compiler infrastructure” *ACM Workshop on Interpreters, Virtual Machines and Emulators,* 2003.

[99] C. Williams and J. Hollingsworth, “Bug driven bug finders”, *International Workshop on Mining Software Repositories*, 2004.

[100]R. Wismuller, “Debugging of globally optimized programs using data flow analysis”, *ACM Conf. on Programming Language Design and Implementation,* 1994.

[101]E. Witchel and M. Rosenblum, “Embra: Fast and flexible machine simulation”, *Conf. on Measurement and Modeling of Computer Systems*, May 1996.

[102]L. Wu, R. Mirani, H. Patil, B. Olsen and W. Hwu, “A new framework for debugging globally optimized code”, *Conf. on Programming Language Design and Implementation,* 1999.

[103]M. Xu, R. Bodik, and M. Hill, “A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay”, *Int’l. Symp. on Computer Architecture*, 2003.

[104]K. Yeung, P. Kelly and S. Bennett, “Dynamic instrumentation for Java using a virtual JVM”, *Performance Analysis and Grid Computing,* 2004.

[105]P. Zellweger, “An interactive high-level debugger for control-flow optimized programs”, *ACM Software Engineering Symposium on High-Level Debugging, pages 159-171,* 1983.

[106]L. Zhang and C. Krintz, “Adaptive code unloading for resource-constrained JVMs”, *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems,* 2004.

[107]S. Zhou, N. Kumar and B. Childers, “Profile guided management of code partitions for embedded systems”, *Design Automation and Test in Europe Conference*, 2004.