Tdb: A Source-level Debugger for Dynamically Translated Programs

Naveen Kumar[†], Bruce R. Childers[†], and Mary Lou Soffa[‡]

[†]Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
{naveen, childers}@cs.pitt.edu

[‡]Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
soffa@cs.virginia.edu

Abstract

Debugging techniques have evolved over the years in response to changes in programming languages, implementation techniques, and user needs. A new type of implementation vehicle for software has emerged that, once again, requires new debugging techniques. Software dynamic translation (SDT) has received much attention due to compelling applications of the technology, including software security checking, binary translation, and dynamic optimization. Using SDT, program code changes dynamically, and thus, debugging techniques developed for statically generated code cannot be used to debug these applications. In this paper, we describe a new debug architecture for applications executing with SDT systems. The architecture provides features that create the illusion that the source program is being debugged, while allowing the SDT system to modify the executing code. We incorporated this architecture in a new tool, called tdb, that integrates a SDT system, Strata, with a widely used debugger, gdb. We evaluated tdb in the context of a code security checker. The results show that a dynamically translated program can be debugged at the source level and that the approach does not overly increase the run-time performance or memory demands of the debugger.

Categories and Subject Descriptors

D.2.5. [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.3.3. [**Programming Languages**]: Language Constructs and Features—*Program instrumentation*, *run-time environments*

General Terms

Languages, Performance, Algorithms

Keywords

Debugging, Dynamic Binary Translation, Dynamic Instrumentation

1 Introduction

Although the importance of debugging techniques to assist users in finding software bugs has long been recognized, new debugging techniques continue to be needed as programming languages, user demands, and implementation strategies change. The goal of a debugger is to accurately respond to user queries and commands from the viewpoint of the source program. Debugging activities include requesting variable values, single stepping through program execution, watching for particular conditions and requests to add and remove breakpoints. In order to respond, the debugger has to map the values and statements that the user **expects** using the source program viewpoint, to the **actual** values and locations of the statements as found in the executable program.

As programming languages have evolved, new debugging techniques have been developed. For example, checkpointing and time stamping techniques have been developed for languages with concurrent constructs [6,19,30]. The pervasive use of code optimizations to improve performance has necessitated techniques that can respond to queries even though the optimization may have changed the number of statement instances and the order of execution [15,25,29].

Currently, a new implementation vehicle, software dynamic translation (SDT), is being increasingly used for important applications, including software security [17,22], dynamic code optimization [1,2,4], binary translation of one instruction set to another [10,11,12,23], host machine virtualization [28], and computer architecture simulation [8,28]. A SDT system translates (generates) target code fragments/traces dynamically and stores the generated code in a fragment cache from which it executes. "Trampoline" code is put in the translated code to switch between the fragment cache and the dynamic translator for code generation. A SDT system may also insert instrumentation code into a translated program to gather run-time information and monitor program behavior.

The motivation to debug dynamically translated programs arises from two sources. First, some environments may have a dynamic translator tightly integrated with the host machine such that all applications must be translated before execution. Therefore, a developer may not have the luxury of debugging the application without dynamic translation. Security systems enforcing security policies by checking an application's binary code fall in this category [5]. Secondly, SDT systems may result in code transformations that expose bugs that are not manifested in the absence of the transformation. An example in this category is a dynamic optimizer [1,2,4].

SDT creates problems in debugging that cannot be solved by current techniques. In particular, debuggers targeted for statically generated code produce debug information at compile-time. The debug information consists of mappings that can be used to relate source code and data to the executable. Since the target code is generated dynamically, this static debug information for relating the untranslated code statements and data values to the executing code, is insufficient. Another problem is that a translated statement can be modified during execution, or a statement may be translated several times such that the number of statement instances can change throughout execution. Yet another problem that current techniques cannot handle is a user may put a breakpoint, using the source program, in code that has not been translated yet. Solving these problems requires techniques that can relate the source code to the executing code, as the code is dynamically generated and modified.

In SDT systems, the trampoline code, as well as instrumentation code, is executed as part of the application code. However, this code should be transparent to the user debugging the application. Similarly, the translation actions should also be hidden from debug users. Therefore, debugging techniques need to be developed that hide execution of everything that is unrelated to source code.

Finally, due to both the growing importance of SDT and the difficulty of developing SDT systems, several frameworks, including Strata [21], Dynamo/RIO [4], and Walkabout [7], have attempted to enable dynamic translator reconfigurability and retargetability to ease the burden of SDT development. As a result, the SDT system can easily be targeted to a different SDT application or machine platform. Hence, there is a strong need to decouple the debugger as much from the SDT system as possible, so that debugging capabilities can be quickly realized for a new SDT system.

In this paper, we present a new debug architecture for SDT systems that achieves transparency while supporting the usual source-level debug queries including step, watch, and set and remove breakpoints. The debug architecture also provides necessary decoupling between the SDT system and the debugger.

This paper provides a reference implementation of the debug architecture. The dynamic code changes that we consider result from basic translations, overhead reduction transformations, and dynamic instrumentation. The basic dynamic translations include generating a new instruction, inserting multiple instructions for a single program statement during translation, ignoring and not generating instructions for a program statement, deletion (flushing) of previously translated instructions, and the duplication of program instructions in the translated code. We also consider several overhead reduction transformations, including instruction trace formation, conditional branch linking, indirect branch translation caching, partial inlining of unconditional branches and calls, and fast return handling [21]. Finally, we consider the effect of insertion and removal of instrumentation in the translated code.

To demonstrate the effectiveness of our techniques, we developed a new debugger, ***tdb***, based on our debug architecture. *Tdb* extends the GNU *gdb* debugger and supports all of *gdb*'s source-level commands and queries. We integrated the SDT framework Strata [21] into the debug architecture and evaluated the debugging capabilities in the context of a software security checker.

This paper makes a number of contributions, including:

- A debug architecture that enables source-level debugging for dynamically translated programs and an implementation of this architecture in a new tool, *tdb*;
- An interface that enables SDT debugging tools to be easily written without regard to machine specific details and SDT system specific details;
- An interface that enables existing debuggers to be easily extended to support dynamically translated programs;
- Debug techniques that provide source-level transparency even though instrumentation and trampoline code are executed as part of the application code;
- Debug techniques that handle the dynamic transformations that are used to reduce translation overhead by avoiding context switches; and,
- A verification and experimental evaluation that shows our approach works well and has minimal overhead.

In the next section, we describe background and the challenges of debugging dynamically translated programs. Section 3 describes an overview of our debug architecture. Section 4 describes the components and interfaces in the debug architecture, including techniques for generating and using debug information, for a basic SDT system. Section 5 describes the generation of debug information for a more complex SDT system. Validation and experimental results of tdb are presented in Section 6. Finally, Section 7 describes related work and Section 8 summarizes and concludes with future work.

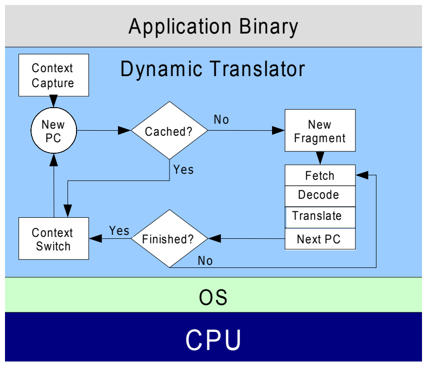


Figure 1: A software dynamic translator

2 Background

Debuggers use breakpoints to enable a user to understand and control a program, including pausing and stepping through execution and inspecting and modifying data values. To answer source-level queries, a debugger maps a program's source code to its binary executable instructions and data locations using information provided by the compiler. A debugger's job is difficult when programs are modified dynamically because information provided by the compiler can become inconsistent with the code modifications.

A SDT is an execution environment that changes code dynamically. A basic SDT system, shown in Figure 1, is a software layer between the application and the underlying operating system. Thus, given an application's source program, a compiler first generates some type of code, including binary or a higher level representation such as byte code. This code, which we call "untranslated" code, is then translated by the SDT to a binary executable. Translated instructions are held in a software cache, called the *fragment cache*. The cache holds "code fragments" (or instruction traces), which are instruction sequences that execute sequentially, ending with a conditional or indirect control transfer.

The working of a SDT system is shown in Figure 1. A SDT system first captures the context of an application (e.g., PC, registers and condition codes) and saves it. Following context capture, the SDT system processes the subsequent application code to be executed.The SDT system checks to see whether the code to be executed has been previously translated and cached. If yes, a context switch restores the application context and begins executing translated application instructions on the host CPU. If the code is not cached, the instructions are fetched and translated one-by-one. When an instruction is translated, it is placed in a fragment within the fragment cache. Instructions are translated until an end of fragment condition is encountered (e.g., a branch or an indirect jump). A control transfer that terminates a fragment returns control to the SDT layer through a trampoline because the code at the control transfer's target may not have been translated yet. When translation of a fragment is complete, the SDT layer performs a context switch to the code cache for execution. Thus, the dynamic translator gets control after each fragment executes to translate subsequent fragments.

Because the SDT layer executes different code than the untranslated code, the source-to-binary debug mappings maintained with static information are insufficient by themselves for debugging translated code. As an example, consider a simple translation that copies a binary instruction from the untranslated code to exactly one location in the fragment cache. We assume that the mappings from source to untranslated code are available from the compiler. During a debug session, when a user places a breakpoint at a *source* program instruction, the debugger cannot locate the *translated* instruction where the breakpoint should actually be placed because of the translation.

One complexity of debugging an application executed with SDT involves the translation of a branch. A branch is handled with a trampoline that is used to re-invoke the dynamic translator to translate the branch's actual target fragment(s). In a debug session, the location of the source branch is mapped to the untranslated branch; however, the executable branch instruction has changed which can affect the reportability of debug queries. When debugging a dynamically modified program with a traditional debugger, the user has to be aware of the trampoline code and the subsequent entry into the translator. Thus, if a user issues a "next" command, he/she has to be aware that the stopping point may be the trampoline code.

Another problem occurs when the translator deletes a instruction, reorders instructions, or introduces more instances of an instruction than were in the untranslated code. Also, the number of instruction instances may change during execution. In these cases, the debugger has to detect the correct instruction instance being executed. Another case happens when the dynamic translator flushes the fragment cache to free space. The debugger needs to recognize that the debug information for the flushed instructions are no longer in effect.

Even more challenging is when the translator applies overhead reduction transformations. Most dynamic translators use several overhead reduction transformations to avoid context switches between the dynamic translator and the code cache, including fragment linking, indirect branch translation caching and chaining, instruction trace formation, fast return handling, and partial inlining of target blocks for unconditional control transfers [2,4,11,21].

Fragment linking rewrites conditional branches to avoid invoking the dynamic translator when the code at the actual target of a branch is already translated. Linking two fragments requires rewriting the existing branch to directly transfer control to the translated target (thus, by-passing trampoline code). If the debugger is unaware of this code transformation, then it may report incorrect information about the translated branch or the trampoline code. Similarly, instructions are emitted to minimize the context-switches into the SDT system from fragments with indirect branches [2,21].

Another overhead reduction technique forms instruction traces, chains of code fragments, to improve cache locality and simplify branch handling. An instruction trace can be, for example, a sequence of instructions on a hot path. Instruction traces present a challenge for a debugger because the traces change at run-time and incur code duplication, code flushes from the fragment cache, branch inversions and fragment linking. The debugger must update its knowledge of duplication, flushes, and dynamic linking and keep these modifications transparent from the user. The other overhead reduction techniques and dynamic instrumentation also perform code modifications that need to be tracked by the debugger.

3 Overview of the SDT Debug Architecture

In this section, we describe a debug architecture that can be used for source-level debugging of dynamically translated programs. The structure of this architecture is shown in Figure 2. The debug architecture has three components: a SDT system, a native debugger and a debug engine. The arrows in the figure represent the invocation of one component (or an action in a repository) by another component. For example, the arrow between the native debugger and the mapper denotes initiation of operations in the mapper.

The debug engine is the key component of the debug architecture, which computes debug information at run-time. This information is used to hide code generation and modification in the SDT system from the native debugger.

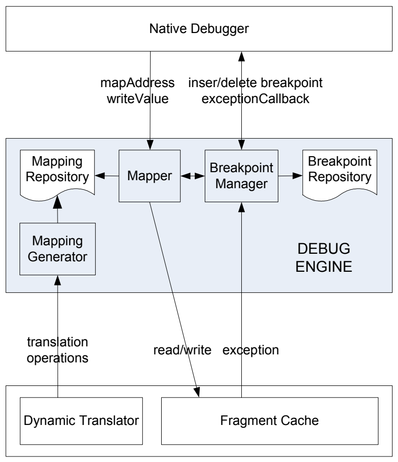


Figure 2: SDT Debug Architecture

The native debugger is similar to a conventional debugger, except that it performs debug actions by targeting the debug engine, rather than the executing program. For example, if a user sets a breakpoint at a source location, the native debugger computes the corresponding untranslated location (binary or bytecode) and notifies the debug engine to set a breakpoint at that location. This is in contrast to the traditional scenario in which the debugger would itself set the breakpoint.

The SDT system is minimally altered to communicate the translation operations to the debug engine. The architecture does not impose restrictions on the SDT system, such as transformations that should not be performed in a debug session. The application being debugged needs to be compiled with the static debug flag turned on, as is traditionally done to debug at source level. The debug architecture does not require any other changes to theprogram.

Debug Engine. The debug engine consists of three components: a *mapping generator*, a *mapper,* and a *breakpoint manager*, and two repositories, the *mapping repository* and the *breakpoint repository.* The components and repositories are shown in Figure 2.

The mapping generator computes debug information, consisting of dynamic debug mappings, to relate source program locations to translated program locations and the vice-versa. It uses information provided by the SDT system for generating and updating mappings. The mappings are stored in the mapping repository.

The mapper uses the debug information from the generator to map untranslated and translated code. The output of the mappings can be used by either the native debugger or the SDT system. The mapper also interacts with the breakpoint manager to determine if a breakpoint needs to be placed in freshly translated code.

The breakpoint manager keeps track of all active breakpoints (and watchpoints) for the executing program in the breakpoint repository. The native debugger communicates information about the breakpoints to the breakpoint manager. The breakpoint manager is responsible for inserting breakpoints in translated code. When breakpoints are hit in the translated code, the breakpoint manager is notified.

To understand the flow of information through the debug engine, consider a typical debug session when a user tries to insert a breakpoint at a source location. The native debugger computes the corresponding untranslated program location and invokes the breakpoint manager. The breakpoint manager consults the mapper to determine the corresponding translated locations and inserts breakpoints at each of these locations. The breakpoint manager also saves the breakpoint information in the breakpoint repository.

When a breakpoint is hit in translated code, an exception is raised which is handled by the breakpoint manager. This handler overrides the exception handler in the native debugger. The breakpoint manager invokes the mapper, which then uses the mapping repository to determine corresponding untranslated location. The untranslated location is passed to the native debugger with the information that a breakpoint has just been hit at that location.

4 The Debug Engine

This section describes how the debug engine orchestrates information flow between the native debugger and the SDT system. We first discuss how mappings are generated. Then, we discuss how the mapper and the breakpoint manager use the mappings to enable debug queries and actions.

4.1 Generating Debug Mappings

The mapping generator uses information received from the SDT system to compute debug mappings. It stores the mappings in the mapping repository. The debug architecture does not specify how the mappings are stored in the repository.

The communication from the SDT system to the mapping generator is performed through a set of API's, shown in Table 1. SDT systems may perform a number of different transformations on the generated code. For example, a dynamic optimizer may reorder statements [2]. The number of transformation operations performed during dynamic translation is diverse and depends on the purpose of translation, such as optimization, security checks, or binary translation, among others. While describing the API's for all of these operations is out of scope for this paper, we discuss the API's for operations performed by a basic SDT system.

A basic SDT system generates code, one fragment at a time (as shown in Figure 1). There are five basic translation operations that are done in a SDT system. Other operations can be handled similarly or using a combination of these operations.

A *regular* operation corresponds to the translation of one instruction to another. An API, as shown in the first row of Table 1, is used to describe this operation. In this API, *u* is the untranslated location, *t* is the translated location, the operation type is *regular*, and the transformation is applied on *code*, as opposed to data.

operation	API
regular	regular(Loc u, Loc t, Trans code)
delete	delete(Loc u, Loc t, Trans code)
many	many(Loc u, Loc t, Trans code)
flush	flush(Loc t, Trans code)
trampoline	trampoline(Loc u, Loc t_1, Loc t_2, Trans code)

Table 1: API's for different translation operations

A *delete* operation happens when an untranslated instruction does not result in any translated code (i.e., the untranslated instruction is "deleted" because it is eliminated in the fragment cache). The second row in Table 1 shows an API that describes this operation. In this API, *u*, represents untranslated location, *t* represents the translated location for the next instruction to be translated (e.g., statement location $u+1$), the operation is *delete*, and the transformation is applied to *code.*

A *many* operation involves an untranslated instruction that is translated into multiple new instructions. An API for this operation is shown in the third row of Table 1, which describes the case when translation of a statement leads to generation of two statements. In this example, *u* represents the untranslated location and *t* is the location for the additional statement.

A *flush* operation evicts translated instructions from the fragment cache. The API shown in the fourth row of Table 1 can be used to describe a flush operation, in which *t* is a translated location containing the statement being flushed.

Finally, a *trampoline* operation handles control transfers that result in trampoline code that transfers control from one location to another. The last row of Table 1 shows an API for a trampoline operation, in which *u* is the untranslated branch location and t_1 and t_2 are the not-taken and taken trampoline locations.

Operations that affect the location of data-values, such as those resulting from statement reordering in a dynamic optimizer, can be described in a similar way. Such operations are not shown in this paper, although the debug architecture described above can be used to generate debug information for these operations.

4.1.1 Dynamic Debug Mappings

Before we describe the generation and update of mappings, we discuss the different categories of mappings that are used for debugging applications translated by a basic SDT system. For brevity, we do not make a distinction between an instruction and its location and use the term "instruction" to refer to its "instruction location" in the following discussion. There are four types of debug mappings:

- *Untranslated-to-translated code mapping* (U-T): This mapping relates untranslated code to translated code. It is bidirectional; i.e., given an untranslated instruction, a corresponding translated instruction can be found, and given a translated instruction, a corresponding untranslated instruction can be found. A U-T mapping is notationally shown as $u \leftrightarrow D$, where *u* is the untranslated statement location and *D* is a set of translated locations.
- *Translated-to-translated code mapping* (T-T): This type relates translated instructions to translated instructions. It is used when a dynamic code modification results in several new instructions for one translated instruction. A T-T mapping is unidirectional and shown as $t_1 \rightarrow \{t_2\}$, where t_1 and t_2 are translated instructions.
- *Translated-to-untranslated code mapping* (T-U): This type relates a translated instruction to an instruction in the untranslated code. It is used to map instructions in the fragment cache that do not have a direct relationship to untranslated instructions (e.g., as in trampoline code). This mapping is unidirectional and shown as $t \rightarrow \{u\}$, where *t* is a translated location and *u* is a location in the untranslated code.

Initially, before an untranslated statement, *u*, is translated, it has the default U-T mapping, $u \leftrightarrow \varnothing$. During translation, an instruction's U-T mappings are updated and T-T and T-U mappings are generated/updated. An update modifies the destinations of a mapping. The dynamic debug mappings are generated/updated differently based on how the code is dynamically transformed. For example, a translation of an instruction into another instruction results in update of a single U-T mapping, while the translation of an instruction to two instructions causes the update of a U-T mapping and the generation of a new T-T mapping. Side effects that result from translation, such as a fragment cache flush also generate and update mappings.

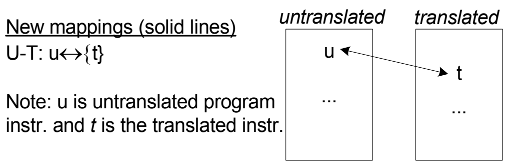
4.1.2 Basic Translation Operations

The mapping algorithms for the five basic translation operations, shown in Table 1, are described below. For each operation, an algorithm is used to update and generate the dynamic mappings. In the algorithms, a mapping for a program or translated instruction is represented as $<s>$, where $<s>=s \rightarrow R$ (or $<s>= s \leftrightarrow R$), *s* is an instruction and *R* is a set of destination instruction(s). For notational convenience, "*" is the set of destinations for a mapping (e.g., if $<s>=s \leftrightarrow R$ in a U-T mapping, then $*=R$). The symbol *u* is an untranslated instruction location and the symbol *t* is a location in the fragment cache. The normal set operators, such as union and difference, can be applied to the mappings.

Regular. For this operation, the mapping generator updates a U-T mapping for a program instruction as shown in Figure 3(a). Consider an untranslated location *u* that is translated to a new location *t*. In this case, the U-T mapping <*u*> is updated to include the new destination *t* in *u*'s set of translated destinations (*). Figure 3(b) shows this mapping between the program instruction *u* and translated location *t*[1].

```
regular(Loc u, Loc t, Trans code) {
    <u> = u↔* ∪{t}
}
```

(a) Algorithm


untranslated
translated
New mappings (solid lines)
U-T: u↔{t}
Note: u is untranslated program instr. and t is the translated instr.
u
...
t
...

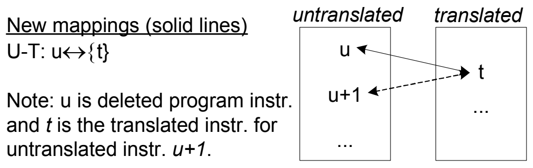
(b) Example

Figure 3: Regular Operation

Delete. The delete operation is shown in Figure 4(a). In this case, the U-T mapping <*u*> for a deleted program instruction *u* is updated to map it to the next translated location *t*. The translation of the next logical program instruction will also be mapped to *t*. Hence, when a program instruction is deleted, the mapping generator maps it to the translated location of the following program instruction as shown in Figure 4(b). Here, *u* is deleted and is mapped to the translated instruction, *t*, for *u+1*.

```
delete(Loc u, Loc t, Trans code) {
    // t is the next translated location and
    // is different than t in regular oper.
    <u> = u↔* ∪ {t}
}
```

(a) Algorithm


untranslated
translated
New mappings (solid lines)
U-T: u↔{t}
Note: u is deleted program instr. and t is the translated instr. for untranslated instr. u+1.
u
u+1
...
t
...

(b) Example

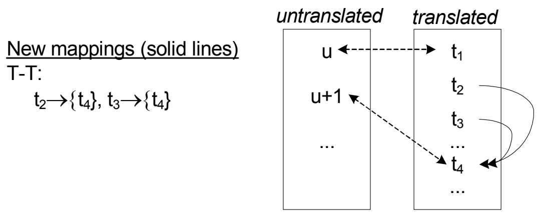
Figure 4: Delete Operation

Many. A translation that produces more than one instruction uses the many operation in Figure 5(a). A T-T mapping is generated for each instruction in the translated sequence that does not have a corresponding location in the untranslated code. For example, if an untranslated instruction at location *u* is translated into instructions at locations t_1, t_2, and t_3, then a T-T mapping is generated for t_2 and t_3 with `many(u, t2)` and `many(u, t3)`. A T-T mapping is not generated for t_1 because the operation *`regular(u, t1)`* is used to update *u*'s mapping to have location t_1 as a destination. In effect, many maps the additional translated instructions t_2 and t_3 to the translated location of the next program instruction. If the next instruction is *u+1*, then t_2 and t_3 have mappings $t_2 \rightarrow$*{translated(u+1)}* and $t_3 \rightarrow$*{translated(u+1)},* where *translated()* gives the translated location of an untranslated program address. Figure 5(b) shows this case where t_2 and t_3 have been mapped to translated instruction, t_4, for the next logical program instruction, *u+1*.

1. In the figures, a solid line shows the new mappings that result from the mapping algorithms.

```
many(Loc u, Loc t, Trans code) {
    // t is an additional instruction; e.g.,
    // t2 or t3 in the example (b)
    if (instruction(u) is not a branch)
        <t> = t→{translated(u+1)}
    else
        <t> = t→{translatedTargets(u)}
}
```

(a) Algorithm



(b) Example

Figure 5: Many Operation

Because *u* may be a branch, the translated instructions may have two possible next logical program instructions (i.e., the taken and not taken targets of the branch at *u*). Hence, when *u* is a branch with taken and not-taken targets, *u+1* and *u+2*, then t_2 and t_3 have mappings $t_2 \rightarrow$*{translated(u+1),translated(u+2)}* and $t_3 \rightarrow$*{translated(u+1),translated(u+2)}*. In the algorithm, the many operation uses `translatedTargets()` to get the set with the taken and not taken translated locations.

The algorithm can be easily extended to the case when the branch targets are not yet translated: they can be handled by updating the mappings when a target block is actually translated (in a way similar to backpatching, except the mappings are patched). The many operation can also be implemented using T-U mappings, however it is more efficient to use the T-T mappings.

Flush. When a translated instruction is removed from the fragment cache (due to cache overflow, self-modifying code, or dynamically linked libraries), the flush operation in Figure 6 is used. When an instruction at translated location *t* is deleted, all mappings with *t* as a source or destination are modified. If *t* is in the set of destinations for a U-T or T-T mapping, then the destination sets are updated by removing *t*. For mappings with *t* as a source, the destination set is set to ∅, effectively deleting the mapping <*t*>.

```
flush(Loc t, Trans code) {
    ∀<u> | t = destination(<u>)
        <u> = u→* - {t}
    ∀<t1> | t = destination(<t1>)
        <t1> = t1→* - {t}
    <t> = t→∅
}
```

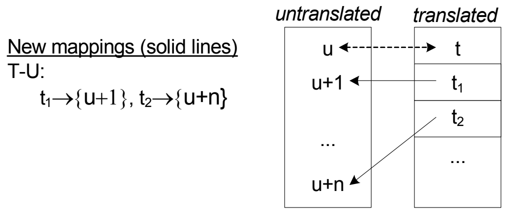
Figure 6: Flush Operation

Trampoline. The trampoline operation in Figure 7(a) is used to generate T-U mappings for each instruction in a trampoline. For a location *t* in the trampoline, a mapping is generated from *t* to the trampoline's target (i.e., the untranslated code). In the algorithm, t_1 is a not-taken trampoline location and t_2 is a taken trampoline location. If the branch is unconditional, then only one trampoline will be generated and only one mapping will be formed.

An example for the not-taken and taken trampolines is shown in Figure 7(b). Here, instruction *u* is the untranslated branch and *t* is the branch at the end of the fragment. When the branch is not taken, control falls through to the not-taken trampoline at t_1. Hence, there is a mapping from t_1 to *u +1*, which is the fall-through location in the untranslated program. Similarly, when the branch at *u* is taken, control is transferred to t_2, which is mapped to *u+n* (the taken target of *u*). If the trampoline consists of several instructions, mappings will be generated for all of them using the trampoline operator.

```
trampoline(Loc u, Loc t1, Loc t2, Trans code){
    <t1> = t1→{notTaken(u)}
    <t2> = t2→{taken(u)}
}
```

(a) Algorithm



(b) Example

Figure 7: Trampoline Operation

4.2 Using Mappings for Debug Queries

The native debugger communicates debug queries and actions to the debug engine. The API's are shown in Table 2. The debug engine uses mappings from the mapping repository to implement these queries and actions on the translated code.

A *write* operation is used by the native debugger to write a value into the translated program. All writes in the native debugger are overloaded in our debug architecture to instead call the *write* API. The API is shown in the first row of Table 2. In this API, *u* represents the untranslated location to be written to; *v* represents the value to be written; and *s* represents the size of data. Note that the location *u* may be a memory location as well as a register.

The *mapAddress* operation is used by the native debugger to request a translated location for an untranslated location. Any debug action using untranslated locations in the native debugger are overloaded in our debug architecture to call the API shown in the second row of Table 2. The address *u* in the API call is the untranslated location.

The *insertBreakpoint* and *removeBreakpoint* operations are used to insert and remove breakpoints in the translated code. The API's for inserting and removing breakpoints are shown in the last two rows of Table 2. In the API's, *u* represents an untranslated location and *t* represents the type of breakpoint (e.g., breakpoint or watchpoint).

operation	API
write	write(Address u, Value v, Size s)
mapAddress	mapAddress(Address u)
insertBreakpoint	insertBreakpoint(Address u, Type t)
removeBreakpoint	removeBreakpoint(Address u, Type t)

Table 2: API's for different debug operations

For each of the operations described above, the debug engine uses mappings in the mapping repository to determine code associations and implement debug actions. We now describe how debug queries and actions can be implemented with our debug architecture.

Set Breakpoint. A user can place a breakpoint at a particular source statement or a function call. To find where the breakpoint should be placed, a traditional debugger uses the program's static symbol information and maps the source statement to the generated (untranslated) code. When the breakpoint is hit at run-time, control is transferred to the debugger, which suspends the program's execution. The user can then set/delete breakpoints, modify program state, single step, or continue execution.

For dynamically translated code, the native debugger maps source statements to untranslated code, using the U-T mappings, and passes this information to the breakpoint manager. The breakpoint manager stores the breakpoint information in the breakpoint repository and invokes the mapper to determine corresponding translated locations. The breakpoint manager then inserts breakpoints at each of the translated locations.

When new code is translated by the SDT system, mappings are generated and stored in the mapping repository. The mapper passes the information about newly generated code to the breakpoint manager. The breakpoint manager looks up the breakpoint repository to determine if breakpoints need to be placed in freshly translated code. If so, the breakpoint manager inserts requisite breakpoints.

When a breakpoint is hit and an exception is raised in the translated code, it is handled by the breakpoint manager. The breakpoint manager uses the mapper to determine the corresponding untranslated location for the breakpoint location. This untranslated location is then passed to the native debugger for further debug actions.

Remove Breakpoint. This command removes a breakpoint. The native debugger calls the breakpoint manager. The breakpoint manager consults the mapper to determine if corresponding translated locations exist for the breakpoint. If so, the breakpoints are removed from all these locations. The breakpoint manager also updates the breakpoint repository.

Continue. A continue command resumes execution after a breakpoint is hit. On a continue command, the debugger puts the untranslated statement in the breakpoint location. It then executes the statement, re-inserts the breakpoint and continues normal execution. With dynamic translation, the same set of actions is taken, except the translated instruction is used by the mapper when the debugger is stopped at a translated location.

Next and Step. A next or step command causes execution to continue until the next source statement is encountered. With hardware-assisted single-step, both commands continue execution from the current breakpoint until a location is reached that corresponds to a different source statement. When the next command is used in dynamically translated code, the debugger can continue execution until a translated code location is reached that can be mapped to a different source statement, using the U-T mappings.

If during the single step command, a translated code location is reached that has a T-T or T-U mapping, then the mapper looks up dynamic mappings and instructs the breakpoint manager to insert a user-invisible breakpoint at the destinations of the mapping. User-invisible breakpoints are used by debuggers to silently step through code (i.e., without notifying the user that a breakpoint has been hit). Execution continues and when the user-invisible breakpoint is hit, the mapper checks to see if the next source statement has been encountered. Hence, the user is unaware that additional code for a many or trampoline has been executed.

Without hardware-assisted single-step, next or step set a breakpoint at the next source statement or all targets of a branch instruction. In this case, next and step are break commands and can be handled similarly to break.

Watchpoint. A watchpoint monitors a change in a program value . If there is hardware assistance for a watchpoint, an exception is raised when the value in the "watched" location changes. This exception is caught by the debugger and the user is notified. In our architecture, this exception is caught by the breakpoint manager. With a dynamic translator, the exception may be raised during the execution of the translator itself or in additional code inserted by the translator (trampoline code, overhead reduction code, or dynamically instrumented code). If a value changes in such a situation, then the mappings are used to determine whether execution should be stopped. In this case, the mapper can resume execution invisibly to the user (and the native debugger) to avoid reporting a change that should be transparent. Without hardware assistance, the debugger single steps until the value at the watched location changes. The mappings can be used to avoid the translator code and the additional code.

Other commands. Other commands such as, disassembly, examining contents of a memory address or a register, setting a value in memory or register, next instruction, step instruction, disable, post-mortem crash analysis, etc. can be implemented using a combination of the techniques above.

5 Overhead Reduction Transformations and Dynamic Instrumentation

Our reference implementation of the debug architecture targets a SDT system that performs basic translation, applies overhead reduction transformations, and performs dynamic instrumentation. In this section, we describe how the overhead reduction transformations and dynamic insertion/removal of instrumentation can be handled. We focus on the generation of mappings by the mapping generator.

5.1 Overhead Reduction Transformations

Dynamic code translations can be more complex than described in the last section, particularly for overhead reduction transformations. For example, an indirect branch may be translated into a series of instructions to perform a lookup in an indirect branch translation cache (IBTC) using the target address. When an IBTC is used, T-T mappings are needed for the IBTC in addition to the U-T mapping for the indirect branch in the untranslated code. A more complex transformation, such as instruction trace formation, involves a number of translations that affect the mappings. It uses the regular, many, flush and trampoline operations to generate its mappings. We describe the mapping algorithms for these two overhead reduction transformations next. Mappings for other overhead reduction techniques can be generated similarly.

```
IBTC(Loc u,Loc t){
    delete(u, t)
    ∀r ∈ (tramp(u)) do
        trampoline(u, ∅, <r>)
}
```

Figure 8: IBTC Translation

IBTC. When an IBTC is used, an indirect branch is eliminated and additional code is generated for a lookup into the IBTC. The algorithm that generates the mappings for the IBTC is shown in Figure 8. A U-T mapping is generated from the untranslated location *u* of the indirect branch to the next translated location *t* using the delete operation. The IBTC lookup code is essentially an indirect branch trampoline and T-T mappings <*r*> are generated with the trampoline operation with target being a *register location*. When the mappings are used by the debugger, the target of the mapping can be inferred by inspecting the correct register.

```
let head be frag starting a trace
let entry be the fragment at the trace entry
retranslate to create the trace
insert a branch in head to jump to entry
update mappings to reflect the trace:
    ∀<r> ∈ mappings(head) do
        flush(<r>)
    t = branch in head fragment to entry
    e = first instruction in trace entry
    <t> = t→{e}
```

Figure 9: Instruction Trace

Instruction Trace. Figure 9 shows a simplified algorithm for generating the mappings for an instruction trace. When a trace is created, the instructions in fragments that form the trace are retranslated. During retranslation, mappings are generated normally for each instruction. After the trace is formed, the fragment that started the trace (the "head fragment") is changed to have a branch at the top of the fragment that jumps to the trace (the "trace entry"), which in effect turns the head into a trampoline to the trace (for any existing fragments that are linked to the head). Because the untranslated code in the head is dead after the trace is formed, all mappings associated with it are flushed. A T-T mapping is created for the branch from the head to the trace entry.

5.2 Dynamically Instrumented Code

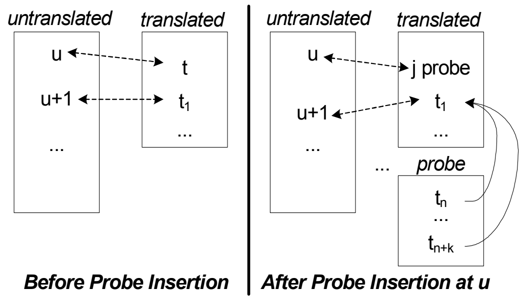
SDT systems typically monitor and gather information about the executing program, and instrumentation code is injected in the translated code for such monitoring and profiling. When debugging a dynamically translated program, the debugger has to ensure that instrumentation code (not part of the program) is transparent to the user. In a way similar to the basic translation operations and the overhead reduction techniques, the debug mappings can be used to hide the instrumentation.

Instrumentation is typically injected with a *probe*. A probe is the code that intercepts program execution to transfer control to a monitor or profiler. Because instrumentation can be *both* inserted and removed on-the-fly, *fast breakpoints* have been used for probes [16]. A fast breakpoint replaces a translated instruction by a jump to a "breakpoint handler" that does the instrumentation. Dynamic instrumentation systems, such as Dyninst [18] use this technique. Note that a fast breakpoint is *not* related to debug breakpoints—a fast breakpoint is inserted and removed for monitoring and profiling, rather than debugging; that is, it is transparent to the user.

When a new probe is inserted, the debug mappings need to be updated to reflect the presence of the probe. Figure 10(a) shows how the mappings are affected. The `insertProbe()` algorithm takes the translated location that is instrumented, *u*, as an input. A many operation is done for each instruction, *r*, in the probe (`probe(u)` returns all locations in the probe at u). That is, a T-U mapping is generated from each instruction in the probe to the next location after *u* (i.e., *u+1*). The U-T mapping from the *u* to its translated location *t* is kept to map the fast breakpoint at *t* to *u*.

```
insertProbe(Loc u, Trans code) {
    ∀<r> ∈ probe(u) do
        many(<u>,<r>)
}
removeProbe(Loc u, Trans code) {
    ∀<r> ∈ probe(u) do
        flush(<r>)
}
```

(a) Algorithms for Probe Insertion and Removal



New mappings (solid lines) after probe insertion
T-T: $t_n \rightarrow t_1$, $t_{n+k} \rightarrow \{t_1\}$

(b) Example
Figure 10: Probe Insertion and Removal

Figure 10(b) show an example before and after a probe is inserted. In the "before case", there is a mapping from program instruction *u* to translated instruction *t* and a mapping from the program instruction *u+1* (the next logical instruction after *u*) to t_1. When a probe is inserted at *t*, a fast breakpoint is inserted to transfer control to the instrumentation code (indicated by `j probe`). Each instruction, t_n ... t_{n+k}, in the instrumentation code is mapped to the next instruction, *u+1,* after *u*. The case when a probe is inlined directly in the translated code can be handled similarly by the algorithm in Figure 10(a). Figure 10(b) shows only a probe that has been inserted with a fast breakpoint.

Because probes may be removed during execution (e.g., when a counter reaches some threshold), the debug mappings should be updated. In this case, all the mappings associated with the instrumentation probe are flushed with the `removeProbe()` algorithm, as shown in Figure 10(a).

Dynamic instrumentation may not always be performed with fast breakpoints. Indeed, the instrumented code can be inlined in the translated code. Our debug techniques can handle such programs, and the algorithms in Figure 10(a) still hold.

6 Evaluation

We developed a debugger, called *tdb*, for dynamically translated programs as a reference implementation of the debug architecture. *Tdb* uses the GNU debugger *gdb* (version 5.3) as the native debugger [24] and supports all source-level commands and queries in *gdb*. The SDT system used to dynamically translate applications was Strata [21]. The translation operations performed by Strata were the five basic translation operations as described in Section 4.1, overhead reduction techniques of fragment linking, IBTC, partial function inlining and instruction traces, and dynamic insertion of probes.

The changes made to gdb involved modifying the functions that read/write program addresses and insert/delete breakpoints to instead target the API's shown in Table 2. The changes made to Strata were also small. The changes included modifying the translation loop of Strata, overhead reduction techniques, and dynamic instrumentation. We used shared memory as a mechanism for communication between different components of the debug engine.

To validate and evaluate *tdb*, we used the debugger in a scenario involving a code security checker. A SDT code security checker instruments instructions in the program binary and performs a security check before executing that instruction. This technique is used in program security applications of SDT such as Dynamo-RIO [17] and Strata [22,21]. Our code security checker is implemented with Strata and can enforce policies on the use of operating system calls, using dynamic instrumentation at system call instructions. For example, the use of file open may be restricted to not open certain files (e.g., the password file). We validated *tdb* to ensure that source-level information can be correctly reported. *Tdb*'s performance and memory overheads for generating and using the mappings were also evaluated.

6.1 Methodology

Several SPEC2000 benchmarks were used to compare the results and overhead of *gdb* and *tdb*. All experiments were run on a 500 MHz Sun Blade 100 with 256 MB RAM and Solaris 9. Strata's default fragment cache size of 2 MB was used and all overhead reduction transformations were enabled. In the security application, all system calls are instrumented. The instrumentation is done with fast breakpoints and enforces the restrictions on operating system services.

We inserted user breakpoints in the benchmarks with *gdb* and *tdb*. 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 statements. 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.

6.2 Verification

To validate the operation of *tdb*, we ensured that it correctly mapped breakpoint locations to appropriate source statements. The validation compared the information reported by *gdb* without dynamic translation to the information reported by *tdb* 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 for *tdb* and *gdb* was also automatically compared.

Table 3 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 statements were hit a total of 1,569 times. We checked whether *gdb* and *tdb* 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. We also verified that the breakpoint commands in both cases reported the same information (e.g., which source line number was hit). Finally, we ensured that the programs ran successfully to completion when all breakpoints were disabled.

6.3 Performance and Memory

To evaluate performance and memory overhead, we compared the run-times of both debuggers and measured the memory requirements of *tdb*. Table 4 shows the run-times for the benchmarks under *gdb* and *tdb* when breakpoints are inserted and hit, according to the methodology in Section 6.1. 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* has run-times from 192 to 379 seconds. *Tdb* 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. We measured where *tdb* spends its time and found that the overhead due to generating and using the mappings is negligible and accounts 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 4 shows in the third and fourth columns ("Total Breakpoints"), *tdb* 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* due to code duplication from fragment overlap (fragments may share code), partial inlining, and instruction traces. *Tdb* inserts breakpoints in the untranslated code and 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* 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* over *gdb* is reasonable, given the large number of breakpoints inserted.

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 3: Number and type of breakpoints hit

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

Table 4: Run-time performance, number of breakpoints, number of mappings, and memory overhead

We also investigated the memory overhead of *tdb,* as shown in the last four columns of Table 4 ("Number of Mappings" and "Memory"). *Tdb*'s memory requirements are related to the size of the mapping and breakpoint tables, with the former dominating. For example, in *mcf*, the number of U-T mappings is 6,081, T-U mappings is 1,701, and T-T mappings is 186. The maximum size of the mapping table is limited by the size of Strata's fragment cache. Entries in the mapping table need 8 bytes for U-T mappings and 4 bytes for the other mappings. In the worst case, there is one entry per instruction in the fragment cache. For a 2 MB fragment 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 Related Work

A number of debuggers for statically optimized code have been developed [3,9,13,14,15,25,29], some of which deal with both data value and code location problems (reordering/deletion of statements). The code location problem in statically optimized code entail similar debug mappings to that proposed in this paper. However, these mappings are generated assuming all of the optimizations have been applied statically. Dynamic mapping must handle having translations done as the code executes as well as having trampoline and instrumentation code. The translation itself also needs to be hidden from the debug users. Jaramillo et al. used mappings, generated staticaly, that are powerful and capture path-sensitive information by analyzing program [15]. However, the mappings require the entire optimized code to be available for mappings. The mapping techniques are also not well-suited to being updated, which is needed for dynamically changing code. Wu et al. used statically generated code location mappings to insert invisible breakpoints and techniques of selective emulation and forward recovery for data reportability [29]. The anchor points described by Wu are similar to our U-T mappings. Tabatabai et al. proposed code location mappings similar to the basic translation mappings proposed in this paper [23,24]. However, these mappings are insufficient for handling the overhead reduction techniques and dynamic instrumentation. Tice et al. proposed changing the source code so that code location mappings better relate the source code and optimized binary [27]. However, a dynamic translator often generates code that has no relation with the source code (e.g., trampolines, instrumentation probe) which would not be handled by this approach. Other work on optimized code has proposed code location mappings [9,13], which are also not suitable in a dynamic translator. There has been much work on the data-value problem in debugging [3,9,13,14,15,25,29]. The data-value problem refers to the challenge of finding the appropriate location (memory or register) of a source-level data construct, when program code is optimized. While this problem is very difficult, a basic dynamic translator does not have this problem (unless dynamic optimizations are applied).

SDT has been used in a variety of scenarios, including architecture simulation [8,28], dynamic optimization [1,2,4], security checking [17,22] and binary translation [10,12,23]. Transmeta's Crusoe and Efficeon processors use code morphing, which performs dynamic binary translation of x86 instructions to VLIW instructions in hardware [10]. In this system, insertion of a breakpoint causes re-translation of the code containing the breakpoint, which raises an exception to transfer control to the debugger. In effect, the debugger modifies the behavior of the translator itself. One of the advantages from a debugger's perspective is that dynamic translation is kept completely transparent to the user. Some other SDT frameworks, such as Dynamo [2] and DynamoRIO [4], provide low-level debug constructs to analyze translated code and data. In these cases, the user must understand the working of the translator and debugging is not transparent. Indeed, this debug mechanism is provided primarily for SDT developers.

The Java Platform Debug Architecture (JPDA) provides an architecture similar to that proposed in this paper [31]. However, the JDPA is very tightly integrated with the underlying just-in-time compiler. This is because, the JDPA targets exactly one Virtual Machine (Java), which is not intended to be reconfigured for different purposes, unlike SDT systems. Further, the JDPA does not provide techniques to generate debug information for SDT systems, where code is generated and modified at a much finer granularity than JIT compilers.

8 Summary

In this paper, we propose a debug architecture that enables source-level debugging of dynamically translated code. We present code mappings that are constructed during dynamic translation and used by the debugger to relate dynamically generated code to source code. We provide a reference implementation of the debug architecture for a retargetable and reconfigurable SDT system, Strata. We also show how to generate mappings for several basic code translations, overhead reduction techniques, and instrumentation probes. Our mappings can be used by a debugger to keep the user unaware of the SDT's existence. We developed a reference implementation of our debug architecture as a new debugger, called *tdb, and* evaluated it. Our experiments show that the debugger has reasonable performance and memory overheads. It is our intention to contribute *tdb* to the public domain for other SDT users and developers.

9 References

[1] M. Arnold, S. Fink, D. Grove, M. Hind and P. Sweeney, "Adaptive optimization in the Jalapeño JVM", *Conf. on Object-Oriented Programming, Systems, Languages and Applications,* 8(10), pages 47–65, 2000.

[2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *Conf. on Programming Language Design and Implementation*, 35(5), pages. 1–12, 2000.

[3] G. Brooks, G. Hansen and S. Simmons, "A new approach to debugging optimized code", *Conf. on Programming Language Design and Implementation*, 27(7), pages 1–11, 1992.

[4] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Int'l. Symp. on Code Generation and Optimization*, pages 265–275, 2003.

[5] D. Bruening, S. Amarsinghe, "Maintaining consistency and bounding capacity of software code caches", *Int'l. Symp. on Code Generation and Optimization*, pages 74–85, 2005.

[6] J. Choi and S. Min, "Race Frontier: Reproducing Data Races in Parallel-Program Debugging", *ACM Symposium on Principles and Practice of Parallel Programming*, 26(7), pages 145–154, 1991.

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

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

[9] M. Copperman, "Debugging optimized code without being Misled", *ACM Transactions on Programming Languages and Systems*, 16(3), pages 387–427, 1994.

[10] 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*, pages 15–24, 2003.

[11] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher, "DELI: A new runtime control point", *Int'l. Symp. on Microarchitecture*, 40(6), pages 201–212, 2002.

[12] K. Ebcioglu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility", *Int'l. Symposium on Computer Architecture*, 25(2), pages 26–37, 1997.

[13] J. Hennessy, "Symbolic debugging of optimized code", *ACM Transactions on Programming Languages and Systems*, 4(3), pages 323–244, 1982.

[14] U. Holzle, C. Chambers and D. Ungar, "Debugging optimized code with dynamic deoptimization", *Conf. on Programming Language Design and Implementation*, 27(7), pages 32–43, 1992.

[15] C. Jaramillo, R. Gupta, and M. L. Soffa, "FULLDOC: A full reporting debugger for optimized code", *Proc. of Static Analysis Symposium*, vol. 1824, pages 240–259, 2000.

[16] P. Kessler, "Fast breakpoints: Design and implementation", *Conf. on Programming Language Design and Implementation*, 25(6), pages 78–84, 1990.

[17] V. Kiriansky, D. Bruening and S. Amarasinghe, "Secure execution via program shepherding", *USENIX Security Symposium*, pages 191–206, 2002.

[18] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, Tia Newhallt, "The Paradyn parallel performance measurement tools", *IEEE Computer*, 8(11), pages 37–46, 1995.

[19] M. Prvulovic and J. Torrellas, "ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes", *Int'l. Symp. on Computer Architecture*, 31(2), pages 110–121, June 2003.

[20] N. Ramsey and D. Hanson, "A retargetable debugger", *Conf. on Programming Language Design and Implementation*, 27(7), pages 22–31, 1992.

[21] 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*, pages 36–47, 2003.

[22] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation", *Annual Computer Security Applications Conference*, page 209, 2002.

[23] A. Srivastava and A. Edwards, "Vulcan: Binary transformation in a distributed environment", Microsoft Research, MSR–TR–2001–50, 2001.

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

[25] A. Tabatabai and T. Gross, "Source-level debugging of scalar optimized code", *Conf. on Programming Language Design and Implementation*, 31(5), pages 33–43, 1996.

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

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

[28] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation", *Conf. on Measurement and Modeling of Computer Systems*, 24(1), pages 68–79, 1996.

[29] 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*, 34(5), pages 181–191, 1999.

[30] M. Xu, R. Bodik, and M. Hill, "A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay", *Int'l. Symp. on Computer Architecture*, 31(2), pages 122–135, 2003.

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