Transparent Debugging of Dynamically Optimized Code

Naveen Kumar
University of Pittsburgh
Pittsburgh, Pennsylvania
naveen@vmware.com

Bruce R. Childers
University of Pittsburgh
Pittsburgh, Pennsylvania
childers@cs.pitt.edu

Mary Lou Soffa
University of Virginia
Charlottesville, Virginia
soffa@virginia.edu

Abstract

Debugging programs at the source level is essential in the software development cycle. With the growing importance of dynamic optimization, there is a clear need for debugging support in the presence of runtime code transformation. This paper presents a framework, called DeDoc, and lightweight techniques that allow debugging at the source level for programs that have been transformed by a trace-based binary dynamic optimizer. Our techniques provide full transparency and hide from the user the effect of dynamic optimizations on code statements and data values. We describe and evaluate an implementation of DeDoc and its techniques that interface a dynamic optimizer with a native debugger. Our experimental results indicate that DeDoc is able to report over 96% of values, that are otherwise not reportable due to code transformations, and incurs less than 1% performance overhead.

1 Introduction

Source-level debugging is the technique of identifying and eliminating program errors, or bugs, using source-level constructs. With the growing complexity of software systems, the importance of debugging continues to be vital to successful software development. Today, support for debugging is expected in any software system including those where code is generated at runtime, e.g., dynamic optimizers.

A dynamic optimizer applies code transformations during program execution based on runtime properties. Dynamic optimization is nearly ubiquitous in JIT-based systems such as Java [2,12] and .NET [4]. Dynamic optimization is also quite popular in the research community, where several prototype systems have been described, including Dynamo [3], Mojo [6], Dynamo-RIO [5], and others [2,16,17]. In each of these systems, dynamic optimization aims to improve program performance. There is another class of software system that imposes instrumentation overheads and uses dynamic optimization to mitigate those overheads. An example of such a system is Pin, where dynamic optimization significantly reduces Pin's dynamic instrumentation overhead [18].

Static and dynamic optimizers perform code transformations, e.g., re-ordering and deletion of statements, that cause the control-flow and data-flow in the optimized code to be inconsistent with the source code. A debugger must relate optimized code with the source code to permit source-level debugging. When optimizations are applied dynamically, the job of a debugger is more difficult than in a static setting. The increased difficulty occurs despite the fact that optimizations performed by dynamic optimizers are often similar to those performed by static optimizers. From a debugging standpoint, what makes dynamic optimization different from static optimization is not the optimizations themselves, but rather the manner in which they are applied. There are several artifacts of dynamic optimization that make source-level debugging more complex, making existing debugging techniques for statically optimized code insufficient:

- Interleaved execution: Dynamic optimizers interleave the execution of the optimized code with optimization passes. A debugger must discern between the optimized program and the optimizer and perform its actions on the program (not on the optimizer).
- Re-optimization: Dynamically optimized code is executed and can later be re-optimized. A debugger must be able to relate the re-optimized code with the source code.
- Dynamic code granularity: Dynamic optimizers often operate at code granularities determined at runtime. For example, a code region that is found to be frequently executed can be a candidate for optimization. A debugger must handle optimizations at any granularity, e.g., individual instructions and data values.
- Unrelated code: Dynamic optimizers often mix additional code with the optimized application binary code. This additional code is unrelated to the unoptimized program and includes instrumentation and control code (to transfer control between the optimizer and optimized code). A debugger must hide the presence of this additional code.

Due to the above complexities, a debugger's job in a dynamic setting is more difficult than simply relating optimized code with unoptimized code. Indeed, few attempts at debugging dynamically optimized code have been made. Systems such as Self obviate the need for debugging optimized code by de-optimizing it when debugging [10]. Java's HotSpot compiler avoids the problem with debugging dynamically optimized by interpreting the unoptimized code during a debug session [12].

Eliminating the need for debugging dynamically optimized code, either using Self's or Java HotSpot's method, or by not debugging dynamically optimized code at all, is not ideal due to three reasons. First, there may be software systems where it is simply not possible to turn off dynamic optimization. A dynamic optimizer in the operating system, or below the operating system, is an example of such a system [5]. Second, optimizations (static or dynamic) are known to expose latent bugs in programs [9]. Therefore, a program may not be fully debugged until debugging is performed with dynamic optimization enabled. Finally, debugging a program in the deployment environment is simply good software engineering practice. In the context of debugging statically optimized code, Hennessy noted in a seminal paper that

"The ability to debug optimized code symbolically and reliably is an important asset that should not be relinquished" [9]. Today, Hennessy's quote is equally relevant to dynamically optimized programs.

Currently, there is need for a debugging solution that can address the complexities associated with dynamically optimized code to permit source-level debugging. The debugging solution must meet several requirements for it to be widely used. First, the solution should be transparent. A user debugging a program should not have to know that the program is dynamically optimized. Second, the solution should be efficient. In a dynamic environment, where a program is modified throughout its execution lifetime, any effort spent in computing information for debugging purposes adds to the overall runtime. The solution must not cause perceptible slowdown. Finally, the solution should be portable. Writing a debugger is a significant investment of time and skills. A solution is desired that can be easily adapted to new architectures and operating systems, as well as new optimizers.

In this paper, we present a debug framework, called DeDoc, applicable to trace-based binary dynamic optimizers[1]. DeDoc is a framework that permits the integration of a dynamic optimizer with a *native debugger*. A native debugger is an existing source-level debugger for binary programs (e.g., gdb). DeDoc enables a debug environment that meets all the challenges and requirements posed by dynamic optimization. This research makes several contributions, including:

- The DeDoc framework: DeDoc consists of techniques to monitor code modifications performed by a dynamic optimizer and generate appropriate information for use by a native debugger. DeDoc's components incorporate these techniques and enhance the capability of native debuggers by adding support for dynamically optimized code.
- Transformation Descriptor: DeDoc introduces the notion of a *transformation descriptor*, which is a property of an instruction or a data value that describes how it was modified during optimization. The transformation descriptors are fine grained to permit DeDoc's techniques to be independent of the granularity at which optimization is applied.
- Debug Engine: A central component of DeDoc, the debug engine, uses the descriptors to generate additional information. The debug engine also integrates with the native debugger to use this information.
- Implementation and Experimental Evaluation: An implementation of the framework that illustrates DeDoc can be used to debug dynamically optimized code at the source level and that its techniques are transparent and efficient.

The rest of this paper is organized as follows. Section 2 gives background necessary to understand our work. Section 3 describes the DeDoc framework. Section 4 details the experimental evaluation. Section 5 presents previous work related to this research. Finally, Section 6 concludes.

1. A trace is a straightline sequence of instructions that can be used as the granularity of dynamic optimization. Most dynamic optimizers operating on binary code are trace-based optimizers [3,5,6,16].

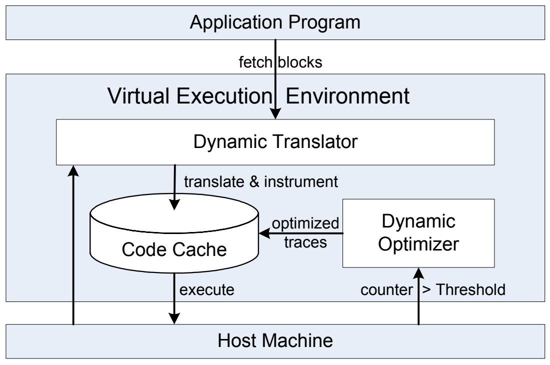


Figure 1: A virtual execution environment for dynamic optimization

2 Background

Source-level debugging involves relating source constructs with their binary counterparts. With program modifications (e.g., optimization), the binary code is not directly related with the source code and data values may be computed earlier or later in the binary code than in the source code. Alternatively, the values may not be computed at all. A debugger must address two problems to permit source-level debugging of optimized code: (1) locating a source statement in optimized code, called the *code location problem*, and (2) extracting the "expected" value of a source variable that is not available because of code modifications, called the *data-value problem* [11]. Debuggers for statically optimized code have solved the code location and data value problems by performing static and dynamic analysis of optimized code to generate *debug information* [1,7,9,11,24]. Debug information, generated during compilation, is used by a debugger to relate optimized code with unoptimized code and answer user queries from the perspective of the source program. When a program is optimized dynamically, the static debug information is inconsistent with the executing program. Furthermore, from the point of view of a debugger, dynamic optimizers perform optimizations in a manner much different than static optimizers, which makes existing techniques for generating static debug information insufficient for dynamic optimizers.

Figure 1 shows the structure and functionality of a (trace-based) dynamic optimizer. A dynamic optimizer is a virtual execution environment that intercepts execution of a program to execute it from a software-managed *code cache*. The *dynamic translator* intercepts the executing program to fetch code blocks one at a time, insert counters and emit the translated blocks into the code cache from where they execute. 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 and optimizes them. Traces are single-entry and multiple exit entities. A trace exit is an "exit stub" that transfers control to either the dynamic translator or other traces.

Figure 2 illustrates the aspects of dynamic optimization that impact source-level debugging. Figure 2(a) shows an example trace with three exit stubs, `e1`, `e2` and `e3`. Execu-

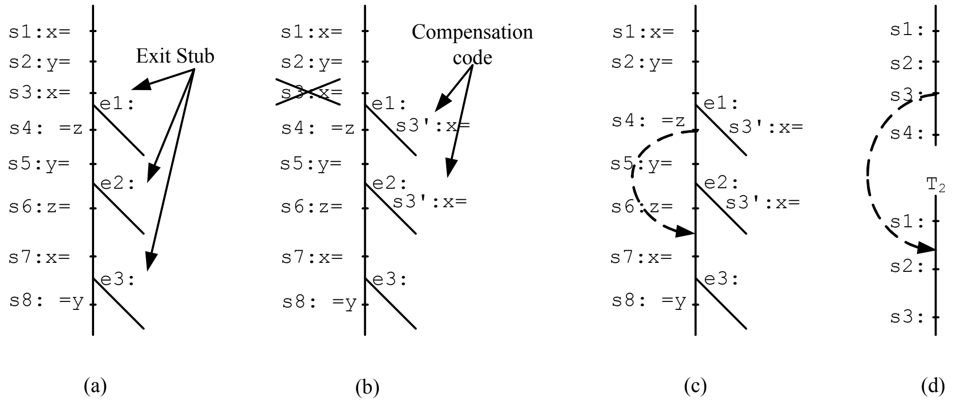


Figure 2: Challenges to debugging optimized instruction traces

tion can reach a trace only at its entry point, i.e., `s1`. Exit stubs transfer control to the dynamic translator for further translation/optimization. Execution of traces is thus interleaved with the execution of the dynamic optimizer. A debugger should allow inspection/modification of program state while traces are executing. It should not allow state inspection/modification when the optimizer executes. Once traces materialize in the code cache, they are linked together, and thereafter, the exit stubs transfer control to other traces.

Figure 2(b) illustrates the effect of an optimization on the trace. The statement `s3` is removed by dead code elimination. Since the optimizer has only a trace view, it must assume that `s3` is live on the path through `e1` and `e2`. Therefore, the optimizer inserts "compensation code" that undoes the effect of dead code elimination in `e1` and `e2`. If a breakpoint is inserted at `s4` in Figure 2(b), the value of `x` will not be the expected one. This data value problem must be addressed by a debugger.

The above data value problem is exacerbated with re-optimization of the trace. Consider the example in Figure 2(c) in which optimizations are applied on the code shown in Figure 2(b). Assume that debug techniques are available such that the expected value of `x` when stopped at `s4` is correctly reported. During re-optimization, `s4` is moved. Debug information generated during re-optimization would relate the code in Figure 2(c) to the code in Figure 2(b). If execution is stopped at `s4`, then the debugger will assume that `x's` value is reportable (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 level of optimization. The challenge in generating debug information during re-optimization is to relate re-optimized code to original code that is no longer available at runtime.

Yet another challenge to debugging is that dynamic optimizers may combine previously optimized traces to perform additional code transformations on the combined trace (a new optimization granularity). Statements from one trace can be moved into another. Figure 2(d) shows an example in which `s3` from trace `T1` is moved to `T2`. Irrespective of the optimization granularity, the debugger must be able to uniquely identify each instruction and data value, that may be queried for, and relate them all the way back to the source code.

Finally, not only the debug information needs to be generated during program execution, it must be communicated for use in debug actions. Furthermore, as traces are deleted and reconstructed, appropriate debug information must be deleted and updated. There needs to be an efficient online mechanism to communicate debug information to the debugger. These challenges are addressed by the DeDoc framework.

3 Debug Framework DeDoc

The primary goal of this research is to keep the dynamic optimizer and its effects transparent from a debug user debugging at the source level. This goal is accomplished by the DeDoc framework. DeDoc, shown in Figure 3(a), has three components: a *trace-based dynamic optimizer (TDO)*, a *debug engine*, and a *native debugger*. In DeDoc, the dynamic optimizer is modified to include a Program Tracker component, which determines programmatic modifications made during optimization. The native debugger is modified

Table 1: The Transformation Descriptors

Transformation Descriptor	Description
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

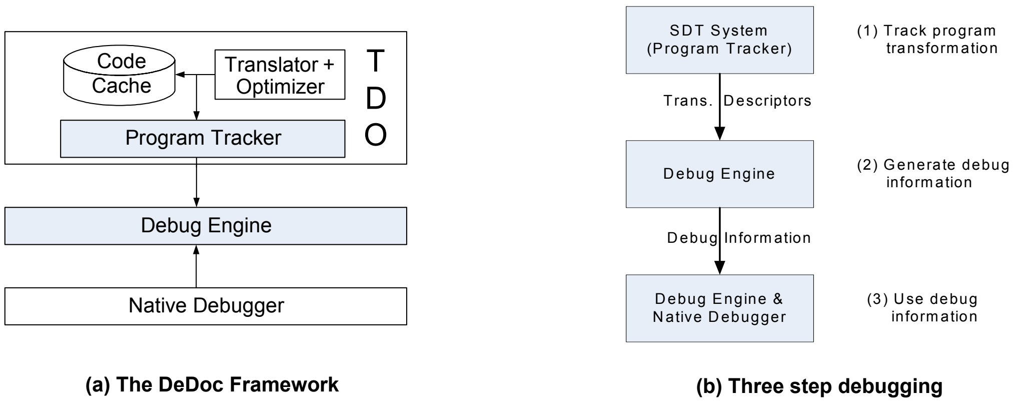


Figure 3: Debugging with the DeDoc framework

to integrate with the debug engine. In essence, DeDoc serves as the "go between" that links the dynamic optimizer and the native debugger. It provides all capabilities to enable source-level debugging of dynamically optimized programs.

With DeDoc, the debug process happens in three steps. Figure 3(b) illustrates the steps. In the first step, the program tracker generates information about the code modifications in the form of *transformation descriptors*. In the second step, the transformation descriptors are used by the debug engine to generate debug information. Debug information is used by the debug engine to hide the effect of program transformations. For example, if a transformation descriptor specifies that a certain data value has been eliminated during dynamic optimization, the corresponding debug information will specify how to determine the deleted value in a debug session. The final step of DeDoc is the use of debug information. DeDoc requires modifications to the native debugger so that its actions on a program are targeted to the debug engine. The debug engine in turn performs the same actions on the dynamically optimized program.

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

3.1 Tracking Program Transformations

A transformation descriptor is an attribute of an instruction or a data value that describes the modifications to an instruction (or data value) from the point of view of the native debugger. Transformation descriptors represent a summary of all modifications to an instruction (or a data value). For example, if a dynamic optimizer applies a set of optimization passes that result in an instruction being moved from its original neighbors, exactly one transformation descriptor is generated to capture the overall code movement.

The motivation for developing and using transformation descriptors is that despite all the differences in what optimizations are performed by a given dynamic optimizer, its transformations can be viewed as a set of basic code edits, including insertion, deletion and movement of code and data values [19]. Transformation descriptors capture these code edits. As a result, DeDoc's use of transformation descriptors eliminates the differences between dynamic optimizers (for expressing transformations) and provides portability across different optimizers. 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. Transformation descriptors, therefore, are a powerful and sufficient technique to describe program transformations performed by dynamic optimizers for the debug operations supported in DeDoc.

There are five 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. Identity is associated with each instruction that is translated but not modified by the dynamic optimizer. CFlush signifies 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.

3.1.1 Generating Transformation Descriptors

DeDoc uses an algorithm, `transprim`, to automatically infer transformation descriptors. `Transprim` is shown in Table 2. `Transprim` deduces the transformation descriptors by comparing the unoptimized trace with its optimized counterpart. Since traces are straightline code sequences, it is possible to detect instructions that have been eliminated or reordered during optimization. `Transprim` requires two preprocessing steps: (1) live ranges of variables are computed before and after register allocation and are available for use; (2) each instruction in the unoptimized trace is assigned a statement-id and its untranslated location is recorded. A statement-id is a unique number associated with an instruction. It is assigned in a linear fashion and remains associated with an instruction even if the instruction is moved. If an instruction is duplicated, all duplicate copies of the instruction have the same statement-id.

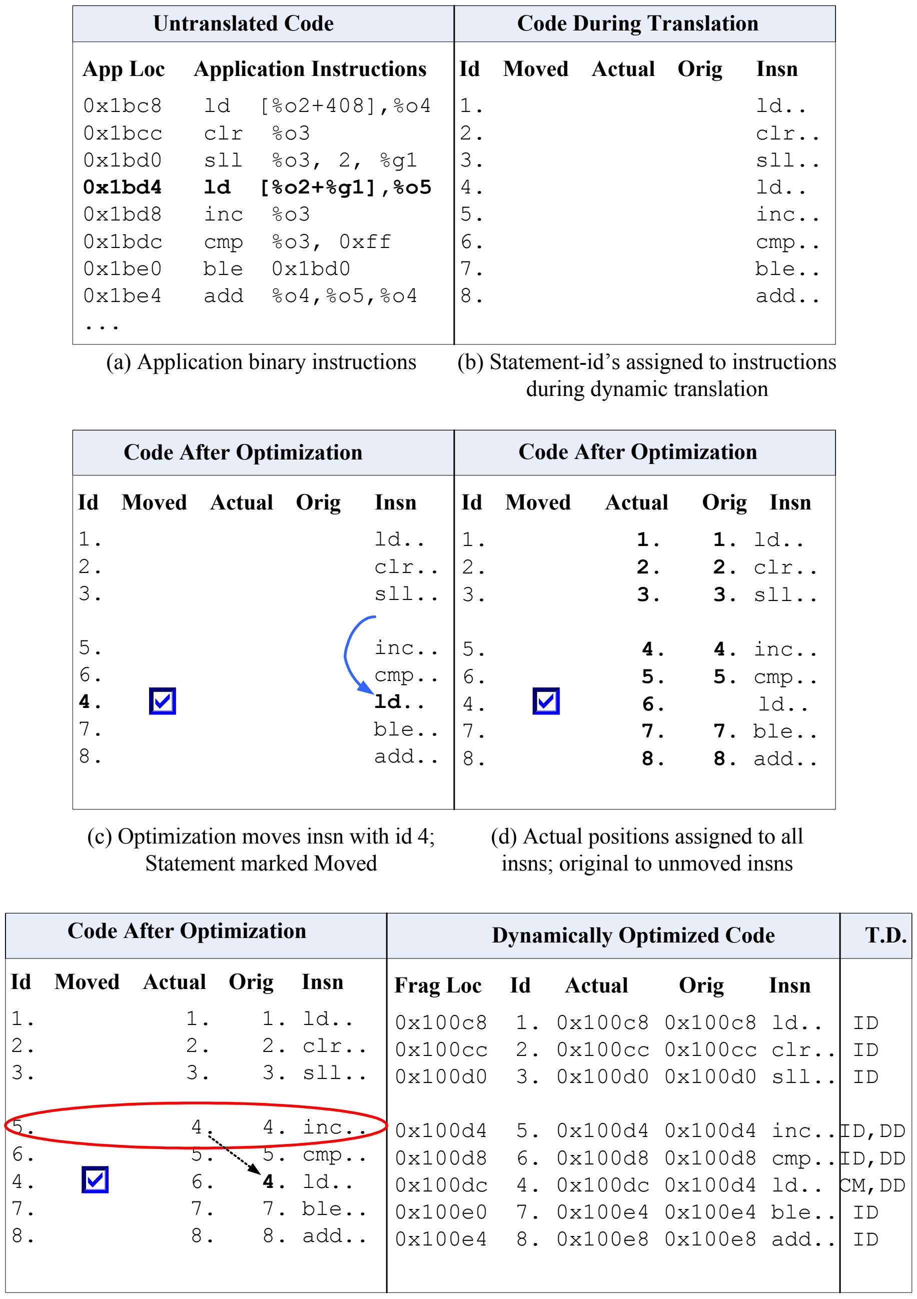
Table 2: Algorithm to generate transformation descriptors for an optimized trace

```
    // 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

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

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

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

12  ∀s : s ∈ Trace // update original positions for all stmts
13     if (s.moved = TRUE) then
14        // find the first instruction on trace with a higher statement-id
15        if ∃s′∈ Trace : (s′.stmtId > s.stmtId) ∧ (s′.moved=FALSE) then
16           s.originalPosition ← s′.actualPosition
17        else
18           s.originalPosition ← ∞
19     else
20        s.originalPosition ← s.actualPosition

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

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

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

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

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

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

`Transprim` is invoked after all optimizations have been applied. When `transprim` is invoked, it first scans instructions in the optimized trace and marks those that have been re-ordered as *moved* (lines 4–7). Note that `transprim` does not accurately detect whether the instruction marked as *moved* was indeed moved—instead, it identifies instructions that are moved with respect to its neighbors, which is sufficient for debugging purposes.

`Transprim` assigns an *actual* position to each instruction in the trace (in a fashion similar to statement-ids), as shown on lines 8–11. An *original* position is subsequently assigned to each instruction, as shown on lines 12–20. Original positions are the same as actual positions for all instructions that have not moved. For instructions that move during optimization, the original position is assigned to be the actual position of the first "unmoved" instruction with a higher statement-id. An original position, intuitively, is the position in the optimized trace where the instruction would have been, had no code movement taken place. Thereafter, transformation descriptors are generated according to Algorithm 1(b) – Algorithm 1(g).

Identity is generated for all "un-moved" instructions, i.e., instructions whose *original* and *actual* positions are the same, *CInsert* for instructions with NULL *original* positions, CDelete for instructions with NULL *actual* positions, *CMove* for instructions whose actual and original positions



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					T.D.
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 4: Algorithm in Table 2 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.

are different and are not NULL, *DDelete* for each instruction where a data value is not available because a computation (instruction) was moved or deleted, and *DMove* is generated for instructions where storage locations of data values are different (e.g., due to register allocation). *DDelete's generation* uses reaching definitions. The generation of *DMove* involves comparing the live ranges of the unoptimized and optimized traces.

3.1.2 Example

Figure 4 uses an example SPARC code snippet to illustrate how `transprim` generates transformation descriptors. The code snippet is shown in Figure 4(a). The first column of Figure 4(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

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

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

the trace is assigned a unique statement-id. The statement-id's are shown in Figure 4(b). In the example, dynamic optimization of the code snippet leads to exactly one code movement, resulting in Identity, CMove and DDelete descriptors. The code movement is depicted by the arrow in Figure 4(c).

`Transprim` marks the instruction with id 4 as moved (Algorithm 1(a)). `Transprim` then assigns actual positions to each instruction. The original position of "unmoved" instructions are set to be the same as their actual positions (see Figure 4(d)). The original position of the moved instruction is set to 4 because it is the actual position of the first "unmoved" instruction with a higher statement-id, i.e., instruction with statement-id 5. The assignment of this original position is shown by the arrow in Figure 4(e).

Once the original and actual positions of all instructions are known, the code transformation descriptors are determined. *Identity* (ID) and *CMove* (CM) are straightforward. *DDelete* (DD) is assigned to instructions with id 5 and 6 because code movement renders the value in *%o5* unavailable at these instructions.

3.2 Generation of Debug Information

Transformation descriptors are used by the debug engine (see Figure 5) to generate *debug information*. Debug information consists of *debug mappings* and *debug plans*. A debug mapping relates code or data value locations in optimized code with those in the unoptimized code. A debug plan relates a code location with a data value storage location and other code locations. Debug plans guide the extraction of runtime variable values that are not reportable due to optimization.

3.2.1 Generation of Debug Mappings

Debug mappings consist of *code location mappings* and *data location mappings*. Code location mappings relates an untranslated or a translated location to another location and helps solve the code location problem. 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`). The mapping relates an untranslated instruction (`headLocation`) with all duplicate copies of the instruction (`TailLocations`) in the code cache.

A data location mapping is also a triple, as shown in the second row of Table 3. A data location mapping relates the location of a data value (`locationBefore`) at a given instruction (`instructionLocation`) with another location (`locationAfter`).

Table 4 shows how the code location and data location mappings are generated for the different descriptors. A code location mapping can be one of three types: `REGULAR`, `DELETE` and `INSERT`. As shown in the first row of Table 4, `REGULAR` mappings are generated by taking the union of all *Identity* and *CMove* descriptors for a given binary location. These mappings are used in a debug session to insert/remove breakpoints in the code cache corresponding to those in unoptimized code.

`DELETE` and `INSERT` are constructed by relating the location in *CDelete* and *CInsert* descriptors with their corresponding postdominators (next instruction) in the code cache. When the native debugger inserts a breakpoint at an instruction with a `DELETE` mapping, the debug engine inserts a breakpoint at the target(s) of the mapping. If such a breakpoint is hit in the code cache during execution, the debug engine reports to the native debugger, the `headLocation` from the `DELETE` mapping as breakpoint location.

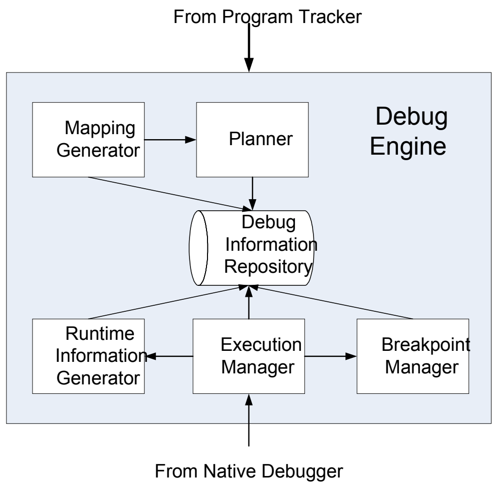


Figure 5: The Debug Engine

`INSERT` mappings are used to hide instructions unrelated to unoptimized code. Execution can pause at an instruction with an `INSERT` mapping, while single-stepping through code. The debug engine's execution manager hides the unrelated instruction by single-stepping until the target of the mapping is reached. Control is returned to the native debugger when the current code cache location does not have an `INSERT` mapping.

For each CFlush descriptor, the mapping generator removes all the associated code location and data location mappings. In addition, the debug engine's planner is invoked so that it can remove the associated debug plans.

DMove is essentially a data location mapping and can be used to relate the storage location of a data value in unoptimized code with that in the optimized code. Each DMove descriptor contains an instruction location, the location of a data value before register allocation and the location after register allocation.

3.2.2 Generation of Debug Plans

The debug engine's planner guides the extraction of runtime data values. While the planner is invoked during dynamic optimization, data value extraction is performed during exe-

Table 4: Algorithms to generate code location and data location mappings

Transformation descriptor	Algorithm: GenerateMappings <descriptor_type>
Identity / CMove	`∀s ∈ ID ∪ CM` **`//instructions with Identity or CMove`**
	`clm ← New(CLM)`
	`clm.type ← REGULAR`
	`clm.headLocation ← s.untransLoc`
	`clm.TailLocations ← {s.cCacheLocation}`
CInsert	`∀s ∈ CI` **`// instructions with CInsert descriptor`**
	`clm ← New(CLM)`
	`clm.type ← INSERT`
	`clm.headLocation ← s.cCacheLocation`
	`clm.TailLocations ← {s.postDominator}`
CDelete	`∀s ∈ CD` **`//instructions with CDelete descriptor`**
	`clm ← New(CLM)`
	`clm.type ← DELETE`
	`clm.headLocation ← s.untransLoc`
	`clm.TailLocations ← {s.postDominator}`
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`

cution by the runtime information generator (see RIG in Figure 5). The planner’s job is to ascertain when and what values need to be extracted.

Consider the example in Figure 4 again. In the figure, a `ld` instruction is moved during dynamic optimization. Suppose, the native debugger needs to report the value in register `%o5` when execution is paused at location `0x100d4` in the dynamically optimized code (see Figure 4(f)). Since the value in register `%o5` is not available until execution reaches `0x100dc` (the new location of the `ld` instruction), a debug plan is generated. The debug plan specifies that when execution reaches `0x100d4`, the debug engine should record all values computed until the instruction at `0x100dc` is executed where the expected value in register `%o5` is known. Thereafter, execution is paused and the debug engine indicates that the unoptimized location corresponding to `0x100d4` has been reached. It reports expected variable values when queried. When execution is continued, instructions are replayed in the expected order. The debug plan for this scenario is:

```
    Debug Plan: <0x100d4, %o5, {0x100dc}>
```

A debug plan includes a *late point*, a data value storage location, and a set of *stop* points. A late point is the same location as the original location of the corresponding moved instruction (e.g., `0x100d4` in Figure 4). Stop points are locations where variables defined by the moved instruction are reachable from the late point (e.g., `0x100dc` in Figure 4). When execution reaches a late point, the debug engine rolls ahead (continues and records) the execution until a stop point is reached. The notion of late and stop points and the technique of rolling ahead execution are borrowed from the Fulldoc debugger [11]. In Fulldoc, the technique of roll-ahead was used in the context of static optimizations.

3.3 Use of Debug Information

Debug information is used by components of the debug engine 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’s components, the execution manager, the breakpoint manager and the RIG, use debug information to take the same action on optimized code in the code cache. In this way, the debug engine hides the dynamic optimizer and its effects (transformations) on a program from the native debugger. As far as the native debugger is concerned, the program being debugged is the unmodified static binary program.

3.3.1 Intercepting the Native Debugger

The Execution Manager is the debug engine’s interface to 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 the program’s address space or insertion/removal of a breakpoint. 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 actions are illustrated in Figure 6. When the native debugger inserts or removes a breakpoint, the execution manager invokes the breakpoint manager. Note that the debug engine may insert its own breakpoints, called *invisible breakpoints,* for maintaining control of the code in the code cache. Examples of invisible breakpoints are breakpoints at late and stop points. Breakpoints that corresponding to native debugger's breakpoints are called *visible breakpoints*.

When the native debugger queries for variable values or the current program counter value (stopped location), the execution manager looks up the DIR and finds alternative locations, if any, to report. When a breakpoint is hit in the program, the execution manager checks with the breakpoint manager to see if it is a visible or an invisible breakpoint. If the breakpoint is visible, the execution manager transfers control to the native debugger for further user queries. If the breakpoint, on the other hand, is a late point, the execution manager invokes the RIG. The RIG records execution of instructions one-by-one until a stop point is reached and then replays the recorded execution in a user-expected manner.

If the native debugger queries the program counter value while single-stepping execution and the current instruction has an `INSERT` mapping, the execution manager invokes the breakpoint manager to insert an invisible breakpoint at the target of the mapping. Execution is subsequently continued until the target is reached. Thereafter, single-stepping is resumed. In this way, the execution manager hides the instrumentation code and exit stubs.

To exemplify the operation of the execution manager, consider Figure 4 again. If the native debugger inserts a breakpoint at location 0x1bd8 in the application binary, the execution manager intercepts this action and consults the breakpoint manager. The breakpoint manager finds that the corresponding code cache location (from a `REGULAR` mapping) is `0x100d4` and inserts a breakpoint at that location. The breakpoint manager also inserts a late point at `0x100d4` and a stop point at `0x100dc`. When execution reaches `0x100d4`, the execution manager invokes the RIG to record execution until `0x100dc` is reached. Once the original `ld` instruction at the stop point is executed, control is returned to the execution manager which is ready to accept further queries from the native debugger.

3.3.2 Breakpoint Handling

The breakpoint manager is a debug engine component that is invoked by the execution manager to insert and remove breakpoints. When the native debugger initiates breakpoint insertion or removal in the application code, the breakpoint manager does the same actions in the code cache. The breakpoint manager uses the `REGULAR` code location mapping of the breakpoint location to determine the corresponding code cache location. When a breakpoint is inserted at a code cache location with a debug plan, the breakpoint manager determines the associated late point and stop points. Invisible breakpoints are inserted at each of these late and stop points.

3.3.3 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 the debug engine component that uses record-replay to extract variable values whose computation have been moved during code transformation. RIG is shown in Figure 5. When a *late* point is reached, the execution manager invokes the RIG and starts the *record phase*.

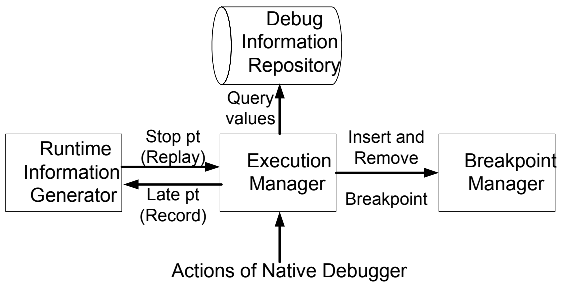


Figure 6: Execution manager intercepts actions of the native debugger and provides transparency

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. Late points encountered during the record phase are also recorded. The record phase continues and when a stop point is hit, the corresponding late point is removed from the list of recorded late points. The *replay phase* starts when no more late points are left. In *replay phase*, breakpoints are reported in the order they were encountered and saved values are reported when queried.

3.3.4 Debug Information Repository

The Debug Information Repository (DIR) is where each debug engine component stores information intended for use by other components. The information stored in the DIR includes mappings, debug plans, values extracted by the RIG and a list of live breakpoints.

4 Implementation and Experiments

We implemented our debug framework and interfaced it with the Strata software dynamic translation system [20] and the widely used Gdb debugger [21]. A dynamic optimizer client, called Strata-DO was implemented. Strata-DO performs the optimizations: constant propagation, copy propagation, redundant load removal, redundancy elimination, partial redundancy elimination, dead code elimination, partial dead code elimination, and loop invariant code motion. It also re-optimizes and combines traces during execution. The implementation is targeted to the SPARC v9 instruction set.

The modifications to gdb include insertion of hooks at different points where gdb performs an action on the program being debugged. Insertion of the hooks required modification to less than *ten* lines of code in gdb. An alternative to using the hooks is to intercept Gdb's calls into the system libraries. Previous work intercepted Linux operating system *ptrace* calls in this way [14]. DeDoc and Strata-DO share some common services, such as intermediate representation construction and manipulation (RTL), which simplified the integration of Strata-DO. By using the common services, less than ten lines in Strata-DO had to be modified to call the Program Tracker.

Table 5: Effect of dynamic optimization on reportability of values

Benchmark	traces	duplicate	debug plans	moved	deleted	non-reportable
mcf	165	64 %	134	2.2 %	0.6 %	2,948
gcc	6,333	60 %	2439	3 %	0.4 %	60,975
gzip	317	65 %	125	1.6 %	1 %	2,250
bzip	356	69 %	241	3 %	0.6 %	2,169
vortex	1,232	58 %	577	0.7 %	0.5 %	8,655
twolf	1,040	61 %	110	2 %	0.15 %	1,210
gap	1,468	58 %	239	2.6 %	0.004 %	1,195

Table 6: Debug-time statistics

Benchmark	#invisible	% breakpoints hit and not reportable	% values not reportable in DeDoc	roll-ahead length
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

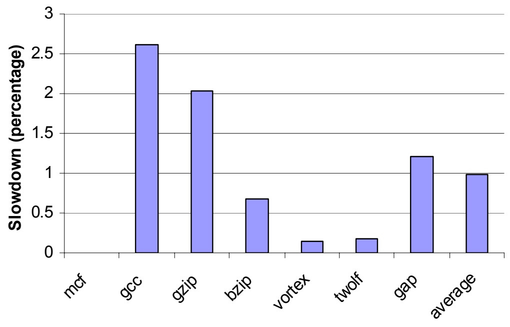
The debug engine is in the address spaces of both the optimizer and Gdb. The components in Strata-DO's address space are: the mapping generator, the planner, the breakpoint manager and the DIR. The execution manager and the record-replay manager are in the address space of Gdb. Calls are made between the components in different address spaces using existing facilities in Gdb.

To determine the effectiveness, in terms of the reportability of values, and efficiency of our debugger, we ran two sets of experiments. The first experiments determined how optimizations affect the reportability of values. The second experiments measured runtime characteristics to determine the performance and memory overheads of DeDoc. For our experiments, we used Strata-DO with a default 4 MB code cache. A Sun Blade 100 system with 256 MB of RAM, running Solaris 9 was used. We used the reference input sets of the SPEC2000 benchmark suite.

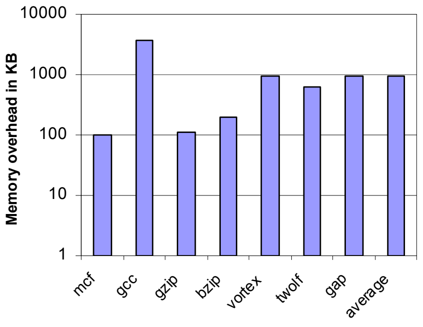
To compute the effects of dynamic optimization on reportability of values, we counted the number of instructions that were moved due to optimization and the variables that were not reportable due to these code movements. We show the results in Table 5. Column 2 gives the total number of traces that were 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, with an average of 552. The 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%. The last column shows the number of variable values that would be not reportable in the absence of DeDoc. The number of not reportable values range from 0.8 to 18 per trace, with an average of 7 non-reportable values per trace. The next set of experiments show how almost all of these values can be reported with DeDoc.

The next experiments gathered the debug-time statistics. For these experiments, breakpoints were inserted at source-level statements 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 the actual run. The inputs to the benchmarks in the training run and the actual run were the same. We selected 50 breakpoints per benchmark. Scripts were used to insert breakpoints and to continue execution until 10,000 breakpoint hits.

The results from the debug-time experiments are shown in Table 6. Column 2 in the table shows the average number of invisible breakpoints inserted per user-visible breakpoint. These breakpoints were 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 optimization, without our framework. The percentage ranges from 5.5% to 97%, with an average of 62%. Although we set breakpoints at instructions where some variables were not reportable, the numbers in this column are less than 100% because instructions duplicated in different traces are often optimized differently. Column 4 shows the percentage of variables at the breakpoints that were not reportable in our framework. The only values not reportable with DeDoc are the ones that are not computed in optimized code. This



(a) Slowdown in generating debug information



(b) Memory overhead

Figure 7: Performance and memory overheads

ranges from 1.98% to 8.4%, with an average of 3.7%. The last column in Table 6 shows the average of roll-ahead in every benchmark due to debug plans. The roll-ahead length ranges from 5 to 25 instructions with an average of 15 instructions. The results demonstrate that even with breakpoints at instructions that have non-reportable variables, DeDoc is able to report 96% of the variables in an expected manner.

We measured the performance and memory impact of generating DeDoc's mappings and debug plans. Figure 7(a) shows the slowdowns in DeDoc for the experimental setup in Table 6. Programs were run with and without generating debug information and the runtimes compared. The slowdown 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. DeDoc's low overhead makes it feasible to generate debug information even when a program is not being debugged. This is useful in analyzing core dumps (post-mortem debugging).

The time taken to hit a breakpoint was also measured and was a constant 0.08 seconds when roll-ahead was not involved. The time taken to roll-ahead one instruction was 0.05 seconds. The actual performance overhead in the debug session varies depending on how often roll-ahead occurs.

Figure 7(b) shows the memory overheads of DeDoc. The memory overhead ranges from 69KB to 2.7 MB, with an average of 685 KB. These overheads include the debug information for traces that are later deleted (e.g., due to code cache flushes). These overheads are comparable to overheads in debuggers for statically optimized code [1,11,24].

From the experiments in this section, we conclude that DeDoc provides complete transparency to native debuggers in the presence of dynamic optimizations. With minimal modifications required to native debuggers and dynamic optimizers, DeDoc hides the effects of optimizations and can accurately respond to user queries even when computations have been re-ordered or eliminated. DeDoc imposes almost no overhead for computing debug information — debug information can be generated for post-mortem debugging even outside of debug sessions. Further, DeDoc's overheads are not perceptible in interactive debug sessions.

5 Related Work

While there is a large body of research work on source-level debugging in general, and source-level debugging of statically optimized code in particular, there has not been any work targeted to dynamic optimization. As mentioned in Section 2, Self and Java's HotSpot compiler have sidestepped the issue of debugging dynamically optimized code by obviating the need for it via dynamic deoptimization and interpretation [10,12].

Most of the previous work related to this research has been in the context of static optimization. The first work was done by Hennessy [9]. Hennessy determined variables whose values are not reportable due to optimizations and the debugger recovered some values to report. In later work, Coutant et al. refined existing techniques to report more variables than done previously. Copperman and Wismuller proposed data-flow analyses to determine which variables are *current* at a statement in statically optimized code [7, 23].

Adl-Tabatabai et al. classify variables by reconstructing the original assignment of variables and report some of those variables [1]. They do not have the code location problem, and the data-value problem is partially handled. Wu et al. base their techniques on Adl-Tabatabai's and Coutant's work and proposed a technique to selectively emulate statements and recover values that could not be reported due to code transformations [24]. Wu used the notions of *interception points* and *anchor points*, which are similar to our *original* and *actual* positions. Wu's work could report even more values than Adl-Tabatabai's, but had some shortcomings.

The Optview debugger uses an interesting approach to debugging where the effects of optimization on code are exposed, rather than hidden [22].

The latest work in debugging optimized code was done by Jaramillo et al. in the debugger Fulldoc. Jaramillo described mappings that could relate every instance of a statement in optimized code with the unoptimized counterpart. Our research uses the *late* and *stop* points developed in Fulldoc.

There has been work on debugging dynamically translated programs. Kumar et al. proposed a debugger Tdb, that provided source-level debugging of dynamically translated code [13]. Tdb does not handle code location and data-value problems posed by code transformations. Tdb uses the technique of inserting hooks into a debugger to hide code location problem from a debugger. DeDoc builds upon Tdb's techniques and hides code location as well as data-value problems from the native debugger.

6 Conclusion

In this paper, we provide a framework, DeDoc, for Debugging Dynamically Optimized Code. DeDoc's approach to debugging is unique: it strives to hide the presence of the dynamic optimizer and its effects on a program's code and data values from the native debugger. DeDoc tracks the effects of dynamic optimizations in terms of transformation descriptors. The transformation descriptors are used to generate debug information. A component of DeDoc, the debug engine, intercepts actions performed by an existing native debugger on a program and uses the debug information to provide a transparent view of the program to the debugger.

DeDoc's techniques are efficient as well as portable. A useful outcome of DeDoc's approach is that it integrates seamlessly with an existing native debugger so that users do not need to learn new commands to debug dynamically optimized programs. We provide an implementation of DeDoc using a dynamic optimizer and a widely used debugger gdb. We also show the performance and memory impacts of our techniques. From our experiments, we notice that DeDoc's techniques can report over 96% variable values that were otherwise non-reportable and incurs under 1% of overhead for computing the required debug information. Our experiments demonstrate that not only dynamically optimized programs can be debugged at source level, but they can be debugged very efficiently.

7 Acknowledgements

Supported in part by NSF grants CCF-0811352, CCF-0811295, CNS-0720483, CCF-0702236, and CNS-0551492.

8 References

[1] A. Adl-Tabatabai and T. Gross, "Source-Level Debugging of Scalar Optimized Code", *Conf. on Programming Language Design and Implementation"*, 1996.

[2] 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,* 2000.

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

[4] D. Box and T. Patiison, "Design and Implementation of Generics for the .Net Common Language Runtime", *ACM SIGPLAN Notices*, 2001

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

[6] W. Chen, S. Lerner, R. Chaiken and D. Gilles, "Mojo: A Dynamic Optimization System", *Workshop on Feedback-Directed and Dynamic Optimization*, 2003.

[7] M. Copperman, "Debugging Optimized Code without being Misled", *Conf. on Programming Language Design and Implementation"*, 1994.

[8] D. Coutant, S. Meloy and M. Ruscetta, "Doc: A Practical Approach to Source-level Debugging of Globally Optimized Code", *Conf. on Programming Language Design and Implementation"*, 1988.

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

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

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

[12] M. Paleczny, C. Vick and C. Click, "The Java HotSpot Server Compiler", *USENIX*, 2001.

[13] N. Kumar, B. Childers and M.L.Soffa, "TDB: A Source-level Debugger for Dynamically Translated Programs", *Symp. on Automated And Analysis-Driven Debugging (AADEBUG)*, 2005.

[14] N. Kumar and R. Peri, "Transparent Debugging of Dynamically Instrumented Programs", *Workshop on Binary Instrumentation and Applications*, 2005.

[15] N. Kumar, "Source Level Debugging of Dynamically Translated Programs", *PhD Thesis, University of Pittsburgh, 2008.*

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation", *Symp. on Code Generation and Optimization*, 2004.

[17] J. Lu, H. Chen, P. Yew, W. Hsu, "Design and Implementation of a Lightweight Dynamic Optimization System", *Journal of Instruction-Level Parallelism*, 2004.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddy and K. Hazelwod, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *Conf. on Programming Language Design and Implementation*, 2005.

[19] 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.

[20] K. Scott, N. Kumar, S. Veluswamy, B. Childers, J. Davidson, M. L. Soffa, "Reconfigurable and retargetable software dynamic translation", *Symp. on Code Generation and Optimization*, 2003.

[21] R. M. Stallman and R. H. Pesch, "Using GDB: A guide to the GNU source-level debugger", GDB version 4.0. Tech. report, Free Software Foundation, Cambridge, MA, 1991.

[22] C. Tice and S. Graham, "OPTVIEW: A New Approach for Examining Optimized Code", *ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, 1998.

[23] R. Wismuller, "Debugging of Globally Optimized Programs using Data Flow Analysis", *Conf. on Programming Language Design and Implementation*, 1994.

[24] 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