Dimension: An Instrumentation Tool for Virtual Execution Environments

Jing Yang Shukang Zhou Mary Lou Soffa

Department of Computer Science, University of Virginia

{jy8y, zhou, soffa}@cs.virginia.edu

ABSTRACT

Translation-based virtual execution environments (VEEs) are becoming increasingly popular because of their usefulness. With dynamic translation, a program in a VEE has two binaries: an input source binary and a dynamically generated target binary. Program analysis is important for these binaries, and both the developers and users of VEEs need an instrumentation system to customize program analysis tools. However, existing instrumentation systems for use in VEEs have two drawbacks. First, they are tightly bound with a specific VEE and thus are difficult to reuse without a lot of effort. Second, most of them can not support instrumentation on both the source and target binaries.

This paper presents Dimension, a flexible tool that provides instrumentation services for a variety of VEEs. To our knowledge, it is the first stand-alone instrumentation tool that is specially designed for use in VEEs. Given an instrumentation specification, Dimension can be used by a VEE to provide customized instrumentation, enabling analyses on both the source and target binaries.

We present two case studies demonstrating that Dimension can be reused easily by different VEEs. We experiment with the two cases and show that the same instrumentation provided by Dimension does not lose efficiency compared to its manual implementation for that particular VEE (the average performance difference is within 2%). We also illustrate that by interfacing with a special VEE that has the same source and target binary formats, Dimension can be used to build an efficient dynamic instrumentation system for traditional execution environments.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – *code generation, compilers, optimization, run-time environments*

General Terms Performance, Design, Experimentation

Keywords Virtual Execution Environment, Dynamic Translation, Instrumentation, Program Analysis Tool

1. INTRODUCTION

Over the last decade, interest in virtual execution environments (VEEs) has been growing with the increased recognition of their usefulness and power. A VEE provides a self-contained operating environment that facilitates programmatic modification of an executing program for diverse purposes, such as architecture-portability [9, 23], performance [2, 3, 14], instrumentation [15, 16, 18], security [19], and power consumption [6]. Many VEEs execute applications by using software dynamic translation, which has the potential to produce high quality code and to utilize resources efficiently. With dynamic translation, a VEE simultaneously handles two different binaries: an input *source binar*y that is typically translated at runtime and a dynamically generated *target binary* that is executed on the host CPU. Our focus in this paper is on translation-based VEEs.

Similar to traditional execution environments, program analysis is an important technique for both the source binary and the target binary of a VEE. Instrumentation, which inserts extra code into a program for profiling, monitoring, and controlling execution, is a widely used technique for enabling program analysis on binary code. Many instrumentation systems have been developed to customize program analysis tools for traditional execution environments. Static instrumentation systems manipulate a statically compiled program before program execution [22]. These systems are unable to handle the target binary in a VEE because the target code is generated on-the-fly during program execution. In contrast, dynamic instrumentation systems directly instrument executable code by deferring instrumentation until runtime [4, 5, 15, 16, 18, 21]. However, existing systems only instrument code that is executable on the host CPU and thus are unable to handle the non-executable source binary in a VEE, such as a Java bytecode program. Some research efforts have been expended on instrumentation in VEEs. For example, Jazz instruments Java programs for structural testing and JRat instruments Java bytecode for runtime analysis [17, 10]. However, these systems are tightly bound with a specific VEE (e.g., Jazz depends on the Java virtual machine) and usually can not instrument both the source and target binaries (e.g., JRat only instruments Java bytecode), limiting their use.

Thus, an instrumentation tool that is stand-alone and can instrument both the source and target binaries would be valuable to both the developers and users of VEEs. VEEs execute on different architectures and are written in different languages. Their source and target binaries have different formats from one VEE to another. What is needed is an instrumentation tool that can be reconfigured to different requirements. In order to develop a stand-alone instrumentation tool for VEEs, details of both instrumentation and virtual execution need to be abstracted.

In this paper, we present Dimension, a tool that provides instrumentation services for VEEs. The objective for Dimension was to build a flexible and efficient instrumentation tool that can be used by a variety of VEEs. Given an instrumentation specification, Dimension can be used by a VEE to provide customized instrumentation, enabling analyses on both the source and target binaries. The design of Dimension identifies the few components of VEEs that need to communicate with an

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.

instrumenter and develops interfaces between the VEE and Dimension. Dimension is designed to be both flexible and comprehensive.

Flexibility. The effort to use Dimension with diverse VEEs is minimal. It is straightforward to modify a VEE to use Dimension, and Dimension can be easily reconfigured to interface with VEEs running on different architectures and written in different programming languages.

Comprehensiveness. Dimension can instrument both the source and target binaries of a VEE. Instrumentation can be done at various levels of granularities from instruction level to method level.

Ease-of-use and efficiency are two other properties that were important in the development of Dimension.

Ease-of-Use. Dimension is transparent to the instrumentation users. Its user-model is similar to ATOM [22] and Pin [15].

Efficiency. Optimization techniques for instrumentation are applied in Dimension with the result that the slowdown from instrumentation is reasonable, compared to other instrumentation tools.

The particular contributions include:

- The first stand-alone instrumentation tool for VEEs that is able to instrument both the source and target binaries at various levels of granularities.
- Simple interfaces between VEEs and Dimension with a VEE providing information readily available.
- Two approaches to reconfigure when Dimension is used for a new architecture or with a new programming language.
- Two case studies and experiments demonstrating the flexibility of using Dimension without losing instrumentation efficiency.
- Extension of the use domain of Dimension to build efficient dynamic instrumentation systems for traditional execution environments.

The remainder of the paper is organized as follows. The next section discusses the design decisions of this work, and section 3 describes Dimension. Section 4 discusses some implementation issues, and Section 5 demonstrates the use of Dimension in two VEEs. Section 6 then evaluates Dimension in both cases. Section 7 discusses future work and Section 8 surveys related work. Finally Section 9 concludes this paper.

2. DESIGN DECISIONS

Although much work has been done on program instrumentation in the past decade, interesting issues arise when we consider instrumentation in a VEE, because of the dynamic nature of the execution environment. The section discusses the design of Dimension, including the paradigm used and its relation with VEEs.

2.1 Paradigm

Three basic approaches to instrument applications in a VEE are illustrated in Figure 1. Paradigm (a) implements instrumentation inside a VEE, intermixing the code for virtual execution and the code for instrumentation. The VEE often needs significant modification, and more importantly, the code for instrumentation is difficult to separate and reuse for other VEEs.

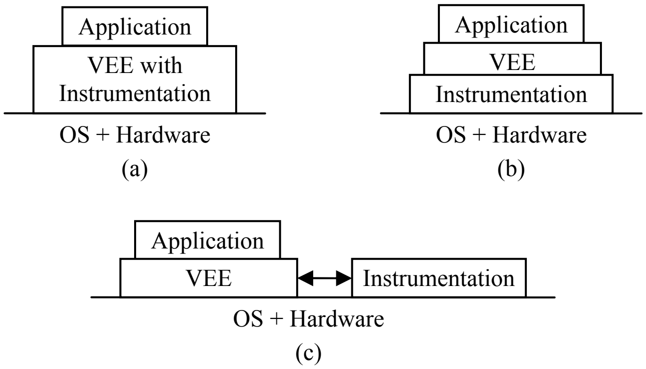


Figure 1. Three possible paradigms to perform instrumentation in a VEE.

Paradigm (b) implements an instrumentation system as another VEE. For example, Pin, a dynamic instrumentation system, is implemented as a translation-based VEE [15]. In this paradigm, a VEE executes applications on top of the instrumentation system. The instrumentation system perceives and translates every instruction executed by the application as well as those by the VEE to add instrumentation during translation. It is able to distinguish application code from the VEE code with hints from the VEE. This paradigm, however, has high runtime overheads. All of the application code needs to be translated by the VEE and then translated again by the instrumentation system before actually being executed on the host CPU. In addition, a VEE's code needs to be translated by the instrumentation system, even if no instrumentation is needed for the VEE code. Extra time and space are consumed by the unnecessary translation. In addition to the extra translation overhead, this paradigm introduces extra context-switches. Every context-switch in normal virtual execution now becomes two context-switches: one is between the application and the VEE, and the other is between the VEE and the instrumentation system.

Paradigm (c) develops the instrumentation system as a separate tool, which provides instrumentation services to VEEs through specific interfaces. Compared to paradigm (a), the separate instrumentation module can be employed by other VEEs. Compared to paradigm (b), the instrumentation module in this paradigm that directly changes the code in VEEs' code cache does not introduce the extra translation and context-switches. Since our goal is to have a separate instrumentation tool that can be used by a variety of VEEs, Dimension uses paradigm (c).

2.2 Instrumentation in VEEs

The relationship with VEEs is critical for the design of Dimension. We briefly review the inherent structure of a typical VEE, and then investigate how virtual execution affects instrumentation. We also discuss our decisions regarding these issues.

As shown in Figure 2, a typical VEE has five conceptual modules: (1) an *initializer* that sets up the entire environment when program execution starts; (2) a *translator* that translates the source binary to the target binary, one code fragment (called a *translation unit*) at a time; (3) a *code cache* that stores the target binary for execution; (4) a *dispatcher* that decides whether the target code of a desired code segment can be directly executed (when it is available in the code cache) or to invoke the translator; and (5) a *finalizer* that cleans up the environment when program execution terminates.

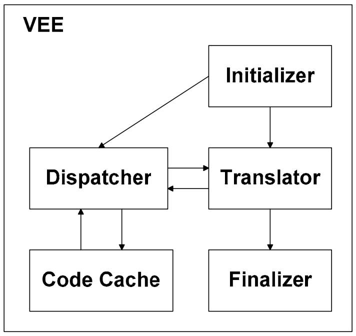


Figure 2. Conceptual modules in a typical VEE.

To avoid interfering with a VEE's code generation and code cache management mechanisms, Dimension uses the *probe-based instrumentation technique*, replacing a program's binary instructions with jumps to invoke the instrumentation code [11]. Therefore, a VEE needs to provide the location of both the source and target code to Dimension, and Dimension then can analyze the instructions and make instrumentation decisions according to an instrumentation specification.

When a user wants source binary instrumentation, one approach actually instruments source binary before translation and the instrumented code is then translated and executed. Extra translation overhead is paid for instrumentation code in this approach. Another scheme modifies the target binary after translation, since the effects of source binary instrumentation can always be achieved by instrumenting the corresponding target binary. This method requires the mapping from the source instructions to the target instructions, and a VEE needs to provide the source-to-target mapping to the instrumentation system. Compared to the first approach, the second method avoids the extra translation overhead, but has more communication overhead. Dimension uses the second approach for simplicity. Therefore, it requires the VEE to provide the source-to-target mapping when source binary instrumentation is needed.

Moreover, on variable-length ISAs, if the VEE uses a multiple-entry translation unit, Dimension has to identify the basic blocks that the unit comprises. On a variable-length ISA, the probe-based instrumentation technique used by Dimension requires that the code unit being instrumented (called an *instrumentation unit*) is single-entry as will be explained in Section 4.1. VEEs' translation units, however, do not always possess this property. The translation units in most modern VEEs fall into four classes: an instruction, a basic block, a superblock, or a method. For the VEEs using an instruction, a basic block, or a superblock as a translation unit, the translation unit is naturally an instrumentation unit as well. For those using a method as a translation unit, the VEEs often have the basic block information of the method for its virtual execution purpose (e.g., Jikes RVM [1]). In our design, therefore, a VEE needs to transmit identification of basic blocks to Dimension, which can use the identification to partition a translation unit into a series of single-entry instrumentation units.

The location of the source and target binaries, the source-to-target mapping, and the identification of basic blocks (if applicable) are what Dimension needs from a VEE. In many cases it is straightforward for a VEE to provide them, because they are either readily available for virtual execution purpose or easily retrieved using existing information.

2.3 Instrumentation Specification

For ease of use, the user-model for Dimension is similar to ATOM [22] and Pin [15]. The user writes an *instrumentation specification* that includes *analysis routines* and *instrumentation routines*. Analysis routines are invoked when execution hits certain program points. Instrumentation routines specify *instrumentation policies* including where to place calls to analysis routines, what arguments are passed, and tasks to be performed at the beginning and the end of program execution. Dimension allows arbitrary analysis routines and provides APIs for users to write instrumentation routines.

```
1  FILE *trace;
2
3  // Called when program begins
4  EXPORT void DIM_ProgramBegin() {
5    trace = fopen("trace.out", "w");
6    DIM_InsertBBCall(SOURCE, ENTRY,
7      FUNCPTR(record_bb), ARG_BB_ADDR, ARG_END);
8  }
9
10 // Called when program ends
11 EXPORT void DIM_ProgramEnd() {
12   fclose(trace);
13 }
14
15 // Print a basic block record
16 void record_bb(void *addr) {
17   fprintf(trace, "%p\n", addr);
18 }
```

Figure 3. Instrumentation specification routines.

We use a basic block tracing tool to demonstrate the ease of using Dimension. Figure 3 is the instrumentation specification that a user would write to print a trace of addresses for every dynamically executed basic block in the source binary in a VEE. Instrumentation routine `DIM_ProgramBegin` (lines 3-8) is a function that is called when the program starts to initialize instrumentation. It opens the `trace` file and uses API `DIM_InsertBBCall` (lines 6 and 7) to inform Dimension a basic-block-level instrumentation policy. This policy is for source binary instrumentation (specified by `SOURCE`) that inserts a call to the analysis routine `record_bb` at the entry (specified by `ENTRY`) of every basic block. The address of the basic block (specified by `ARG_BB_ADDR`) is passed as an argument. `DIM_ProgramEnd` (lines 10-13), another instrumentation routine, is called when the program ends to perform tasks needed to complete instrumentation. Lines 15-18 are the definition of the analysis routine `record_bb`, which prints the address of a basic block to the `trace` file.

Although the example only shows fairly simple instrumentation, Dimension has been used to build various complex program analysis tools, such as hardware cache simulators and program structure testers.

Note that the code works with any VEE that uses instrumentation services from Dimension. The user does not need to know how the target binary is generated, what the granularity of a translation unit is, or how the code cache is managed. In addition, the code also works regardless of which architecture the VEE is running on and which programming language is used to implement the VEE.

3. DIMENSION

This section presents Dimension, which follows the design described in the previous section. We first discuss the component organization of Dimension. We then describe its simple communication interfaces that interact with VEEs. We also discuss Dimension's instrumentation mechanism. We conclude this section by discussing how to reconfigure Dimension when it is used (1) on a new architecture and (2) with a VEE that is implemented in a different language than the one used to implement Dimension.

3.1 Component Organization

We decompose Dimension into two groups of components. One group interfaces with a VEE and the second group is VEE-transparent. Figure 4 shows the component organization of Dimension as well as how components communicate with a VEE.

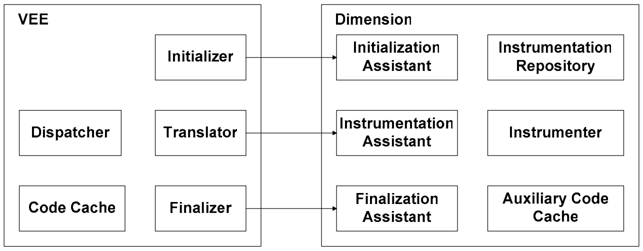


Figure 4. Interfaces between Dimension and a VEE.

3.1.1 Components that Interface with VEEs

Dimension has three components that interface with VEEs. A VEE needs to know how to pass information to employ Dimension, including formats, which turn out to be fairly simple. Note that although the three components are relevant to VEEs, they are VEE-independent, requiring no modification when Dimension is reused with a new VEE.

- *Initialization Assistant.* The execution of the initialization assistant is triggered by a VEE's initializer, as shown in Figure 4. It sets up Dimension at the beginning of program execution. It also loads the instrumentation specification and stores it in Dimension, so that Dimension can instrument programs according to the specification.
- *Instrumentation Assistant*. As shown in Figure 4, the instrumentation assistant receives code segments with relevant information from a VEE's translator. It also prepares for actual instrumentation (the preparation will be further discussed in Section 3.3). The instrumentation assistant is triggered immediately after the VEE translates a translation unit. Starting from the instrumentation assistant, Dimension instruments the code right after translation, which not only guarantees that all dynamically executed code is instrumented, but also handles self-modifying code as long as the code is re-translated by the VEE. Communication at this point also avoids interfering with the VEEs' code generation mechanism.
- *Finalization Assistant*. As Figure 4 shows, the finalization assistant is invoked by a VEE's finalizer. It cleans up Dimension at the end of program execution. It also finalizes instrumentation results, such as writing instrumentation results to a file.

3.1.2 Other Components

Besides the above three components, Dimension has other components that do not interface with the VEE. They include:

- *Instrumenter*. Invoked by the instrumentation assistant, the instrumenter instruments the code received from the instrumentation assistant according to the instrumentation specification.
- *Auxiliary Code Cache*. The auxiliary code cache stores instrumentation code generated by Dimension.
- *Instrumentation Repository*. The instrumentation repository maintains information that is used for instrumentation purposes, such as instrumentation specification and source-to-target mapping.

The instrumenter, in particular, is the only architecture-dependent component, which needs to be modified when the format of the source binary or the target binary is changed.

3.2 Communication Interfaces

Dimension works as an independent module with well-defined interfaces to communicate with VEEs. We minimize the communication interfaces in terms of both the number and complexity. On the other hand, the interfaces have to convey sufficient information so that Dimension is able to accomplish comprehensive instrumentation, such as instrumenting arbitrary instructions in a binary. As Figure 4 shows, three Dimension components (the initialization assistant, the instrumentation assistant, and the finalization assistant) communicate with three VEE modules (the initializer, the translator, and the finalizer) through three interfaces.

3.2.1 Interface `InitDimension`

A VEE calls the `InitDimension` interface to trigger the initialization of Dimension. In particular, as shown in Figure 4, the VEE's initializer uses it to invoke Dimension's initialization assistant. The format is as follows.

```
void InitDimension();
```

This interface is quite simple, requiring no parameters.

3.2.2 Interface `StartInstrumentation`

A VEE calls the `StartInstrumentation` interface to pass a translation unit to Dimension, which then immediately instruments this translation unit. In particular, as shown in Figure 4, the VEE's translator uses it to invoke Dimension's instrumentation assistant. The format is as follows.

```
void StartInstrumentation
        (addr src_start, addr src_end,
         addr tgt_start, addr tgt_end,
         src_to_tgt_mapping map, bb_info bb);
```

`StartInstrumentation` needs six parameters: (1) the start address, (2) the end address of the source binary code of the translation unit, (3) the start address, (4) the end address of the target binary code of the translation unit, (5) the source-to-target mapping information, and (6) the identification of basic blocks in the translation unit. While the first four parameters are always readily available, the other two need more consideration.

Identification of basic blocks is needed only if the VEE executes on a variable-length ISA, and it uses a multiple-entry translation unit. For VEEs that translate from machine code to machine code, the translation unit is usually single-entry (e.g., a single instruction, a basic block, or a super block). For VEEs that translate from some intermediate representation to machine code, the translation unit can be multiple-entry, typically a method or a function. In these cases, information to partition the unit into basic blocks is always maintained by the VEE for virtual execution purpose (e.g., Jikes RVM uses basic block information to perform garbage collection [1]).

Source-to-target mapping information is needed when applying source binary instrumentation. For instructions in the source binary and target binary, the mapping can be many-to-one, one-to-many or one-to-one. If a VEE has not already provided it, the VEE developer needs to add the functionality to support Dimension. A challenge in maintaining the exact mapping information comes from a VEE's code optimization, which may move instructions (e.g., scheduling and partial redundancy elimination) or alter the control flow structure using code replication (e.g., function inlining and loop unrolling). Previous work has shown that for most classical optimizations (e.g., loop transformations), the exact mapping information from un-optimized code to optimized code can be achieved, except dead statements (which apparently need no instrumentation) [7, 8].

3.2.3 Interface `FinalizeDimension`

A VEE calls the `FinalizeDimension` interface to trigger the finalization assistant of Dimension. In particular, as shown in Figure 4, the VEE's finalizer uses it to invoke Dimension's finalization assistant. The format is as follows.

```
void FinalizeDimension();
```

This interface is also quite simple, requiring no parameters.

3.3 Instrumentation Mechanism

With the knowledge of Dimension's component organization and its interfaces with VEEs, we now discuss the instrumentation mechanism, and Figure 5 and 6 show its key algorithms. We start the discussion by addressing target binary instrumentation, followed by source binary instrumentation. We then present several approaches in Dimension to reduce the slowdown of instrumentation.

3.3.1 Target Binary Instrumentation

We emphasize the parts in our algorithms that are relevant to target binary instrumentation in this sub-section. When program execution begins, the VEE's initializer triggers the execution of Dimension's initialization assistant. The initialization assistant loads the instrumentation specification and stores it in the instrumentation repository.

During program execution, every time after the VEE translates a code segment, Dimension's instrumentation assistant is invoked through `StartInstrumentation`, whose pseudo-code is shown in Figure 5. If the translation unit is multiple-entry, line 6 partitions the translation unit into a series of single-entry basic blocks, using the identification of basic blocks provided by the VEE. Lines 8-16 then pass the target binary code to the instrumenter, one single-entry code segment (as an instrumentation unit) at a time.

Figure 6 shows `InstrumentUnit`, which instruments each single-entry unit. The instrumenter scans the target instructions, finds appropriate ones according to each instrumentation policy stored in the instrumentation repository, and then generates a preliminary instrumentation plan (lines 17-25). The plan includes the target instructions to be instrumented, the associated analysis routine, and the arguments to be passed. On line 27, the preliminary plan is optimized for efficiency (details of optimization will be discussed in Section 3.3.3).

```
1  StartInstrumentation(addr src_start,
2      addr src_end, addr tgt_start, addr tgt_end,
3      src_to_tgt_mapping map, bb_info bb) {
4
5    store_mapping(map, repository);
6    bbs[] = partition_bb(src_start, src_end, bb);
7
8    foreach basic block b in bbs[] {
9      <usrc_start, usrc_end> =
10          get_bb_boundary(b, <src_start, src_end>);
11     <utgt_start, utgt_end> =
12          get_bb_boundary(b, <tgt_start, tgt_end>,
13                          map);
14     InstrumentUnit(usrc_start, usrc_end,
15          utgt_start, utgt_end);
16   }
17 }
```

Figure 5. `StartInstrumentation` algorithm.

```
1  InstrumentUnit(addr usrc_start, addr usrc_end,
2      addr utgt_start, addr utgt_end) {
3
4    p = load_policy(repository);
5    if(p needs source instrumentation) {
6      map = load_mapping(repository);
7      foreach source insn si between
8          usrc_start and usrc_end {
9        if(si belongs to p.where) {
10         ti = map_tgt(map, si);
11         record_plan(plan_pool, ti.addr,
12             p.analysis_routine, p.parameter);
13       }
14     }
15   }
16
17   if(p needs target instrumentation) {
18     foreach target insn ti between
19         utgt_start and utgt_end {
20       if(ti belongs to p.where) {
21         record_plan(plan_pool, ti.addr,
22             p.analysis_routine, p.parameter);
23       }
24     }
25   }
26
27   opt_plan_pool = opt_plan(plan_pool);
28   foreach optimized plan op in opt_plan_pool {
29     trampoline = gen_trampoline
30         (op.analysis_routine, op.parameter);
31     replace_jump(op.addr, trampoline);
32   }
33 }
```

Figure 6. `InstrumentUnit` algorithm.

Once an optimized instrumentation plan is generated, Dimension performs actual instrumentation (lines 28-32). As discussed earlier, Dimension does not interfere with VEEs' code cache management. Therefore, it cannot shift or extend code in VEEs' code cache. To instrument a target instruction, the instrumenter replaces it with a jump that branches to a *trampoline*. A trampoline is a code sequence that performs a context-switch, prepares the parameters to be passed to analysis routines, and transfers control to analysis routines. It also executes the replaced instruction. In this way, when execution hits the instrumented location, it follows the trampoline to execute the analysis routines with all parameters and context properly maintained. A jump and its associated trampoline are a probe. The trampolines are stored in Dimension's auxiliary code cache, and thus they do not interfere with VEEs' code cache management.

When program execution terminates, the VEE's finalizer invokes Dimension's finalization assistant. The finalization assistant completes instrumentation and records the result according to the instrumentation specification.

3.3.2 Source Binary Instrumentation

The process of source binary instrumentation is quite similar to that of target binary instrumentation, except that it needs several more steps to locate where to instrument. We emphasize the parts of our algorithms that are exclusively relevant to source binary instrumentation in this sub-section. When Dimension's instrumentation assistant is invoked, it first stores the source-to-target mapping in the instrumentation repository (line 5 in Figure 5), then passes the code (both the source binary and the target binary) to the instrumenter, one single-entry code segment at a time (lines 14-15 in Figure 5). When the instrumenter receives an instrumentation unit, it scans the source instructions, finds appropriate ones according to the instrumentation specification (stored in the instrumentation repository), and uses the source-to-target mapping (also stored in the instrumentation repository) to locate the corresponding target instructions that should be instrumented (lines 7-10 in Figure 6). After this point, source binary instrumentation is the same as target binary instrumentation.

3.3.3 Optimizing Instrumentation

As suggested by [12, 13, 15], most of the slowdown from instrumentation is caused by executing the instrumentation code. Therefore, after a preliminary instrumentation plan is generated, Dimension optimizes it to reduce the overhead of instrumentation code. The optimization techniques are applied automatically in a similar manner as conventional compiler optimization techniques. In probe-based instrumentation technique, the overhead of instrumentation code includes the time (1) to execute the jump which branches to the trampoline, (2) to perform the context-switch, (3) to transfer control to analysis routines, and (4) to execute analysis routines. Dimension uses different techniques that have been shown effective to improve them.

To lower the first overhead, Dimension uses probe coalescing to reduce the number of the jumps that branch to trampolines [12]. For each instrumentation unit (a single-entry code segment), the probe coalescing technique combines separate probes into a single one, keeping exactly the same functionality (i.e., all the analysis routines of the previously separate probes are now invoked in the combined probe's trampoline in the same order). Note that, if arguments need to be passed to the analysis routine, probes can be coalesced only if all the parameter values remain available.

The context-switch overhead can be reduced by partial context-switch. Dimension first analyzes analysis routines to determine a partial context (i.e., registers in most platforms) that may actually be used in analysis routines. During instrumentation, Dimension only generates code to save and to restore this partial context. The overhead to maintain unused context can thus be saved.

Dimension also inlines analysis routines into trampolines to reduce the overhead of transfer of control from trampolines to analysis routines. Currently, we only inline short analysis routines to avoid code expansion.

Finally, for overhead (4), executing analysis routines, Dimension is able to invoke a light-weighted binary-to-binary optimizer to improve analysis routines' code quality at runtime.

As our experiments demonstrated, these optimizations greatly reduce the slowdown of instrumentation. Moreover, these improvement techniques are quite consistent with our objective (a flexible tool that provides instrumentation services to VEEs). Similar to a compiler, Dimension automatically performs these optimization techniques to any instrumentation specified by users, and the application is transparent to the executing VEE or VEE user.

3.4 Reconfiguring Dimension

Dimension has the flexibility to work with a variety of VEEs. However, as for many binary-level systems, it has to be reconfigured to be used. Among its six components, the instrumenter directly handles binaries and thus is architecture-dependent. The initialization assistant, the instrumentation assistant, and the finalization assistant communicate with VEEs, so they are sensitive to the language that implements the VEE's initializer, translator, and finalizer. The other two components (the auxiliary code cache and instrumentation repository) are generic and do not need modification.

3.4.1 For New Architectures

A binary-level instrumentation system is architecture-dependent, because it must understand (decode) and modify (encode) binary instructions. In particular, Dimension must be able to decode both the source and target binaries, and be able to encode the target one. Therefore, we need a new version of Dimension when the format of either the source or the target binary changes.

To facilitate reconfiguration, we develop a *binary-editing utility library* for each expected binary format. A library is associated with a binary format, and can be used regardless of whether the format is the source binary or the target binary in a VEE. Binary-editing utility libraries provide general binary rewriting services to Dimension, such as decoding, encoding, generating, and replacing binary instructions. Dimension's source code uses these utility library calls, instead of directly handling binary instructions. When Dimension needs to be reconfigured to a new architecture, we only need to implement those utility library functions for the new architecture, while the source code of Dimension does not need change, which effectively reduces the reconfiguration efforts.

3.4.2 For New Languages Used in VEE Implementation

Dimension is currently written in the programming language C. C is used to implement many VEEs, and these VEEs can use Dimension through the three communication interfaces (i.e., three functions written in C). However, Dimension needs some reconfiguration if a VEE is written in a language that cannot directly call functions written in C. For example, the translator in Jikes RVM [1] is written in Java that cannot call C functions directly.

Fortunately, most languages provide some mechanism to call C functions. Java, for example, is able to call C functions through Java Native Interface (JNI). When reconfiguring Dimension for a new language, we only need write three *stub functions* using the language's "native interface". The first stub function (`InitDimensionStub`) is to call `InitDimension`; the second one (`StartInstrumentationStub`) wraps information received from the VEE into required parameters and calls `StartInstrumentation`; and the third one (`FinalizeDimensionStub`) is to call `FinalizeDimension`. VEEs written in the new language are able to call the three stub functions, getting through the interfaces indirectly.

We avoid re-writing the interfaces (C functions) by writing new stub functions that are simple and easy to write. The stub function for Java to call `InitDimension`, for instance, is as the following.

```
JNIEXPORT void JNICALL
Java_Initializer_InitDimensionStub(JNIEnv *env,
                                   jobject jobj) {
  InitDimension();
}
```

Note that the number of architectures and programming languages is limited while the number of VEEs is growing. As configurations for new architectures and languages are developed and installed in the libraries, Dimension will be able to use these configurations to instrument new VEEs.

4. IMPLEMENTATION

Dimension is compiled as a shared object to be loaded dynamically by a VEE. It uses about 100 KB address space for code (including the binary-editing utility library) and 3 MB for data (2 MB for the auxiliary code cache and 1 MB for the instrumentation repository). Each instance of Dimension is configured for a specific source binary format and a specific target binary format. In this section, we discuss several implementation challenges that are caused by Dimension's portability to different architectures and VEE implementation languages.

4.1 Probe-Based Instrumentation

As discussed in Section 2.2, Dimension employs the probe-based technique to instrument programs. While the implementation is quite straightforward for fixed-length ISAs, where a jump always replaces a single complete instruction, it is much more difficult for variable-length ones.

In a variable-length ISA, a jump may be longer than the original instruction, so that several continuous instructions need to be replaced. If, however, one of these instructions is the target of a control-transfer instruction, the program may potentially execute an illegal instruction that is part of a jump. This situation places a restriction on the probe-based technique – the instrumentation unit should be single-entry, which is also required by Dimension as described in Section 2.2.

Another situation happens when the instrumentation unit is smaller than the size of a jump. In such cases, we use a shorter instruction instead of jump (e.g., INT 3 in x86) for replacement purpose. Although this short instruction may be more expensive to execute than a regular jump, they are rarely executed since most instrumentation units in programs are longer than a jump.

Since a jump may not always replace complete instructions, another issue in probe-based instrumentation is what instructions should be relocated to the trampoline. Dimension uses a mechanism that is similar to Dyninst [4]. It relocates each complete instruction as long as part of the instruction is replaced by the jump. If the relocated instruction is a PC-relative branch or jump, its offset is re-calculated to match the target address. A relocated call is always emulated as a jump which has the same addressing type. This mechanism is general enough to be used for all variable-length ISAs and does not imply either performance or memory overhead.

In Dimension, if a jump replaces several instructions and more than one instruction need to be instrumented, the corresponding analysis routine for each instrumented instruction is called inside the same trampoline. For example, suppose the x86 code contains three instructions – a two-byte-long integer-add, a two-byte-long integer-sub and a three-byte-long integer-add. The user wants to instrument each integer-add and a five-byte-long jump is used to construct the probe. Figure 7 illustrates how Dimension deals with such a situation – it replaces all three instructions with a jump to the trampoline. The three instructions are executed sequentially, and the analysis routine is called once for each of the integer-adds.

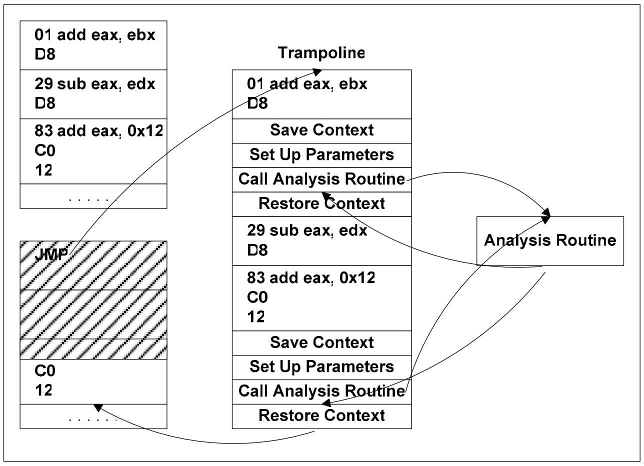


Figure 7. An example of how Dimension handles variable-length ISA.

4.2 Parameter Wrapping in Stub Functions

For a VEE that is written in a programming language that cannot directly call C functions, stub functions have to be invoked to access the interfaces of Dimension (described in Section 3.4.2). Among the three stub functions, `StartInstrumentationStub` has the special functionality to prepare the six parameters required by `StartInstrumentation`.

In Dimension, the choice of the six parameters follows the C language feature, using a start address and an end address to represent a binary code segment. In our implementation, each address is defined as a pointer to a byte (i.e., char *). However, this representation is not appropriate for all programming languages, especially those that do not support pointer types.

As a bridge, `StartInstrumentationStub` has to wrap the information which VEEs provide to the formats that `StartInstrumentation` expects. For example, to write stub function `StartInstrumentationStub` for Java, we use a byte array to represent a binary code unit. This information can be easily provided by a VEE written in Java and is straightforward to be converted to the arguments required by `StartInstrumentation`. `GetByteArrayElements`, a JNI function, is used to get the start address and end address (defined as C byte pointers) from a Java byte array.

5. CASE STUDY

In this section, we describe two case studies of different VEEs that use Dimension, namely, Strata [20] and Jikes RVM [1]. Strata is a reconfigurable and retargetable software dynamic translation framework. In our experiment, we chose the version running on the SPARC V8/V9 ISA and Solaris operating system. Jikes RVM is a multi-platform Java virtual machine, and we chose the version developed for the IA-32 ISA and Linux operating system. We mainly focus on the efforts to insert the interfaces into both VEEs, demonstrating Dimension's flexibility. We also discuss the binary-editing utility library development for both SPARC and IA-32, which contains several interesting issues.

5.1 Dimension-Strata on SPARC/Solaris

Strata is written in C so that Dimension interfaces can be directly invoked. It has an explicit function that initializes the application and another one that is invoked when the application exits. Therefore, we invoke `InitDimension` and `FinalizeDimension` inside these two functions. Similarly, the place where Strata finishes the translation of each code segment is easy to locate, which is also in a single function. The translation unit of Strata is a single-entry fragment containing a straight-line sequence of instructions. Therefore, for `StartInstrumentation`, Strata only has to specify five parameters, and not the basic block information. In Strata, source-to-target instruction mapping is mainly one-to-one, except for some transfer-control instructions whose mapping can be easily obtained.

The SPARC instructions are uniform in length and have a relatively simple encoding. However, SPARC has delay slot instructions (DSI), which are associated with control transfer instructions. DSIs on SPARC can be annulled, i.e., not executed if the branch is taken or not taken. To instrument a DSI, we instrument the control transfer instruction right before it to capture the values in the general-purpose registers. If the control transfer instruction is an annulled branch, we test the condition code before calling the analysis routine. SPARC maintains a register window for twenty-four general-purpose registers, which can be efficiently saved and restored by a single instruction. A partial context-switch applies when the analysis routine does not use the eight global registers, the floating-point state register, the conditional code, or the Y register. The binary-editing utility library for SPARC/Solaris contains 1,500 lines of code taking one person four days to finish.

5.2 Dimension-Jikes RVM on IA-32/Linux

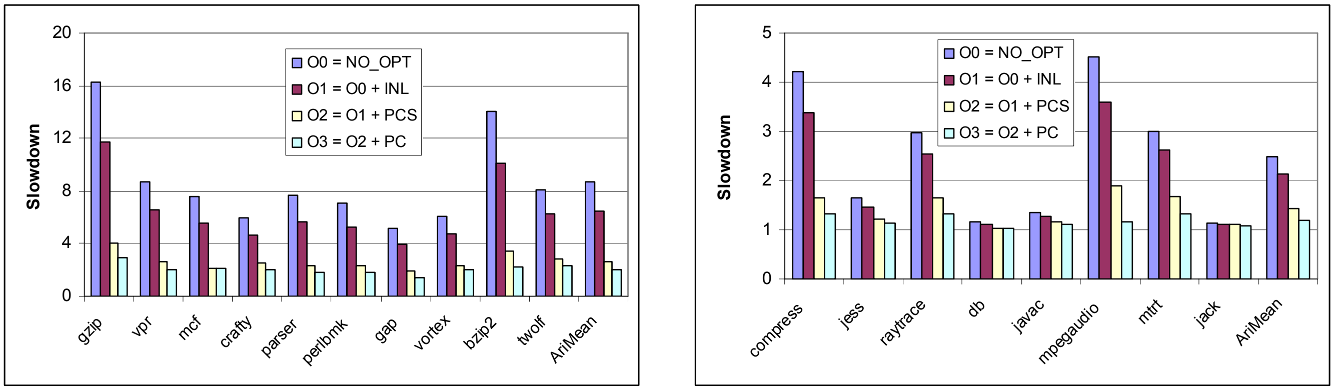
Jikes RVM is mainly written in Java so that it has to call the Java version stub functions. It provides two APIs that insert monitors when the application starts to run and exits. Using these two APIs, we insert `InitDimensionStub` and `FinalizeDimensionStub` into Jikes RVM. Jikes RVM also has an explicit class that compiles the Java bytecode, where we insert the `StartInstrumentationStub`. The translation unit of Jikes RVM is a method, with the basic block information provided. The mapping between bytecode and the compiled machine code is also accessed directly after translation. Both information can be easily extracted from Jikes RVM's data structure (implemented as arrays) and stored as required parameters.

The IA-32 ISA has variable-length instructions, which are difficult to decode and encode. It also has so limited general-purpose registers and spilling is frequently needed. The opportunity of partial context-switch on IA-32 mostly comes from the eflags register, whose access is rather expensive. Two cases can happen here: (1) the analysis routine does not modify eflags so that no save and restore is needed; (2) the analysis routine owns instructions that only modify the lowest byte of eflags (e.g., JA re/8), so that the inexpensive LAHF and SAHF instructions can be used instead of expensive PUSHFD and POPFD. Our lightweight implementation of the binary-editing utility library for IA-32/Linux contains about 2,500 lines of code and took one person 7 days to finish.

6. EXPERIMENTS AND EVALUATION

In this section, we performed experiments to determine the efficiency of Dimension. Three different experiments were performed to evaluate Dimension's performance from different reference points. The first experiment focuses on the instrumentation optimization mechanisms that are described in Section 3.3.3. Each of the optimizations is evaluated to determine its effectiveness to improve Dimension's performance. In the second experiment, the same instrumentation that was manually implemented in the VEE (paradigm (a) in Figure 1) is performed automatically by interfacing with Dimension. By comparing the two performances, Dimension is evaluated to see how its portability feature can affect its performance. In the last experiment, we demonstrate that the use domain of Dimension can be extended to provide efficient instrumentation in traditional execution environments.

The experimental setup is as follows. For Strata, we run SPECint2000 benchmarks (using reference inputs) compiled with gcc -O3 option in the first experiment and cc -O5 option in the third experiment. For Jikes RVM, we run SPECjvm98 (using data sets provided in the benchmark). We choose to use the baseline compiler everywhere and employ mark-and-sweep garbage collection.



(a) Strata

(b) Jikes RVM

Figure 8. Slowdown from target binary instrumentation. (a) is for Strata and (b) is for Jikes RVM. "NO_OPT" stands for no optimization; "INL" stands for inlining; "PCS" stands for partial context-switch; and "PC" stands for probe coalescing.

6.1 Effectiveness of Optimizations

In this experiment, we use a simple instrumentation – calculating the average integer-add instructions executed in each basic block – to show how each of our optimizations can reduce the slowdown from instrumentation. For Jikes RVM, two experiments are performed: one instruments the source binary and the other instruments the target binary. For Strata, only target binary is instrumented.

Figure 8 shows the performance of both VEEs with the target binary instrumentation. To focus on Dimension's performance, the time shown is normalized to the VEE execution time without instrumentation. Each benchmark is executed using four different optimization levels. Without any optimization, the slowdown is fairly large, up to 16.2x in Strata (gzip) and 4.5x in Jikes RVM (mpegaudio). Inlining helps, with the average slowdown improving from 8.6x to 6.4x in Strata and from 2.4x to 2.1x in Jikes RVM. A significant performance improvement comes from partial context-switch, which reduces the average slowdown to 2.6x in Strata and 1.4x in Jikes RVM. Probe coalescing finally reduces it to 2.0x in Strata and 1.1x in Jikes RVM.

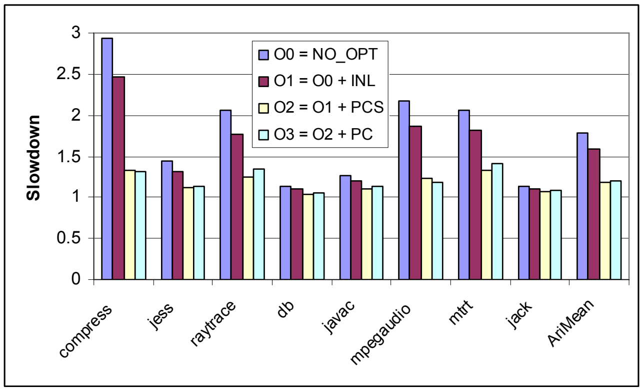


Figure 9. Slowdown from source binary instrumentation in Jikes RVM. "NO_OPT" stands for no optimization; "INL" stands for inlining; "PCS" stands for partial context-switch; and "PC" stands for probe coalescing.

Figure 9 shows the slowdown from source binary instrumentation in Jikes RVM, which follows the same trend (the average slowdown is reduced from 1.7x without optimization to 1.2x after fully optimized), except that probe coalescing actually slightly increases the average slowdown. This is because integer-add is relatively infrequent in Java bytecode compared to that in the target binary, limiting the benefits from probe coalescing but applying it still introduces considerable runtime overhead. Probe coalescing benefits the instrumentation on the target binary of Jikes RVM because there are many opportunities: the number of integer-add instructions in the translated machine code is higher than that in Java bytecode due to translation, e.g., a load and a store in Java bytecode is translated to several target instructions, including an integer-add.

As shown, the optimizations applied in Dimension effectively reduce the slowdown from instrumentation in both the source and target binaries. Their benefits are consistent across the two different VEEs but, as usual, depend on the number of optimization opportunities.

6.2 Efficiency

In this experiment, we investigate how Dimension's flexibility affects the efficiency, i.e., the performance difference between Dimension and instrumentation systems that are built using paradigm (a) in Figure 1. We use Dimension to perform the same instrumentation that has been manually implemented on a VEE and compare their performances.

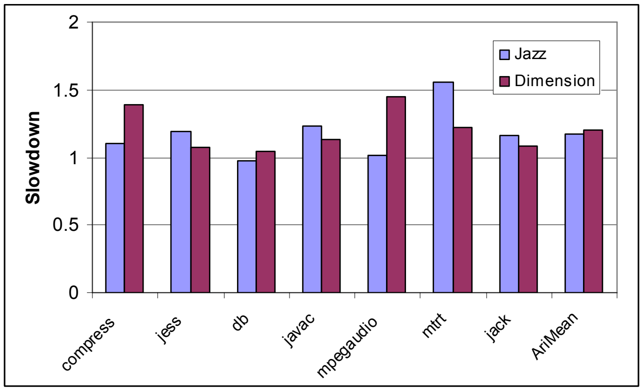


Figure 10. Comparison of slowdown from instrumentation between Jazz and Dimension.

Jazz [17] builds a branch coverage tester by instrumenting each basic block of a Java program to determining what edges are covered by an execution. The system is built on the IA-32 version of Jikes RVM. We use Dimension to build the same instrumentation for Jikes RVM and compare their performances. To our knowledge, Jazz uses all the instrumentation optimization techniques that Dimension uses, so the comparison can fairly illustrate the tradeoff between efficiency and flexibility. Figure 10 shows the slow down caused by instrumentation, which is normalized to the execution time of Jikes RVM without instrumentation. Although for each single benchmark, the performance varies between two implementations, the average slowdown is comparable.

As demonstrated, Dimension achieves comparable performance against the systems in which instrumentation is manually built into a VEE. The flexibility of Dimension introduces negligible effect on its performance.

6.3 Dimension in Traditional Execution Environments

Although Dimension is specially designed for use in VEEs, its use domain easily extends to traditional execution environments. A dynamic instrumentation system can be easily built using Dimension on a VEE. Conceptually, the combination of the VEE and Dimension can be considered as a system that uses virtual execution technique to implement instrumentation, and this approach is used to develop many dynamic instrumentation systems, such as Pin, Diota and Valgrind [15, 16, 18]. The source and target binaries of the VEE should be in the same format.

We built an instrumentation tool (called *Strata-Dimension*) by using Dimension with Strata on SPARC. The process is straightforward and quite easy as discussed previously. Strata is an efficient software dynamic translation framework, with an average slowdown of 1.3x over native execution on SPARC [20]. It translates the source binary to the almost identical target binary, but ensuring it gains the control of the execution. Therefore, Strata is a good choice on which a regular instrumentation system for use in traditional execution environments can be built with Dimension.

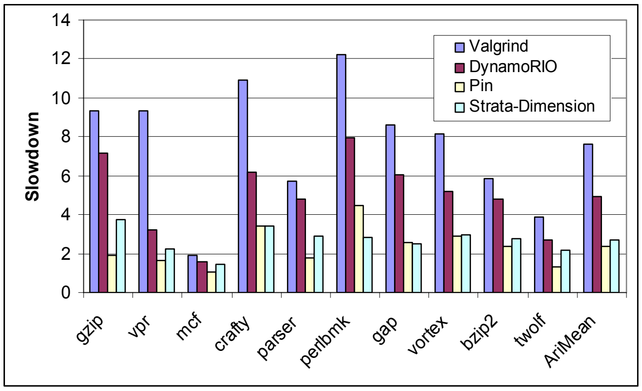


Figure 11. Comparison of slowdown from instrumentation in traditional execution environments.

In this experiment, we use a standard instrumentation, basic-block counting, to compare the performance of Strata-Dimension against Pin, Valgrind, and DynamoRIO. The metric used is the slowdown from instrumentation, which is the execution time with instrumentation normalized to the native execution. The data for Valgrind, DynamoRIO and Pin were obtained from a public publication [15]. As Figure 11 illustrates, Strata-Dimension introduces a reasonable slowdown from instrumentation. The average slowdown for Strata-Dimension is 2.6x, which is slightly worse than Pin (2.3x) but better than both DynamoRIO (4.9x) and Valgrind (7.5x). Pin performs more complicated optimizations on instrumentation than Dimension does. DynamoRIO is primarily designed for dynamic optimization, and does not automatically perform instrumentation optimizations (e.g., partial context-switch) as Dimension and Pin. We believe the slowdown of Valgrind mainly comes from its overall infrastructure, which has a slowdown of 5.6x even when program is executed without instrumentation. The experiment demonstrates that Dimension can be used to build efficient dynamic instrumentation tools in traditional execution environments.

From experimental results, Dimension achieves flexibility and efficiency simultaneously. The optimization techniques applied by Dimension effectively reduce the slowdown from instrumentation, leading to a comparable performance against systems in which instrumentation is developed for a particular VEE. The use domain of Dimension can be easily extended, e.g., building efficient regular instrumentation systems for use in traditional execution environments.

7. FUTURE WORK

Dimension has several limitations that prevent it from being a completely “transparent” tool and seeking the solutions for them will be the focus of our future work.

Firstly, Dimension needs the VEE to provide basic block information or source-to-target mapping when applicable. If the required information cannot be provided by the VEE developer, Dimension loses the ability to do certain instrumentation. A potential solution is that Dimension determines this information by itself. Through careful analysis of both the source binary and the target binary, Dimension may be able to find the correct way to partition a translation unit into single-entry instrumentation units. It may also apply certain de-optimization techniques to map the source and target instructions.

Secondly, the reconfiguration of Dimension needs to write the binary-editing utility library and stub functions if the architecture or language is not in the libraries yet. Currently the implementation is done manually, which is cumbersome, especially when the architecture is very complex. The future work for this is to implement a framework which can generate a new binary-editing utility library and all the new stub functions based on the specifications (e.g., instruction format) of that architecture and language.

Finally, the current Dimension does not capture the high-level context of both binaries, such as finding an arbitrary local variable in a Java bytecode method. By capturing these contexts, Dimension’s power will be increased to implement much more complex program analysis tools.

8. RELATED WORK

At the highest level, binary instrumentation can be classified into two categories: static binary instrumentation rewrites the program before it executes while dynamic binary instrumentation inserts extra code on demand during program execution.

Static binary instrumentation systems were pioneered by ATOM [22]. They have inherent drawbacks in dealing with mixing code and data, indirect branches, shared libraries, dynamically-generated code and self-modifying code.

Dynamic binary instrumentation systems can overcome the above limitations by identifying code at runtime. Many systems use the probe-based approach [4, 5, 21], which works by dynamically replacing instructions in the original program with jumps to the instrumentation code. This technique is based on the fast breakpoint technique proposed by [11]. Many other systems use the jit-based approach [15, 16, 18], which translates the program on-the-fly to directly add the instrumentation code in.

Some VEEs have been manually extended to provide instrumentation. Jazz [17] is a program testing tool which builds instrumentation functionality in Jikes RVM for structural testing. DynamoRIO [3] was extended from a dynamic optimization system, with the addition of a set of instrumentation APIs for building customized program analysis tools. Both Jazz and DynamoRIO provide customized instrumentation by extending the original translation-based VEE. However, designs and implementations are for one particular system, which lacks the portability to a wide variety of different translation-based VEEs.

Our work is highly motivated by FIST [13], which is a binary instrumentation toolkit that is portable to different translation-based VEEs. However, FIST neither tries to minimize the modification to the VEE nor shows convincing evidence that the built-on instrumentation can be both effective and efficient.

Several techniques have been used by previous probe-based instrumentation systems to deal with variable-length instruction sets. DTrace replaces each instrumented instruction with an OS-trap, which is the shortest instruction in some architectures (e.g., x86) [5]. Though only one instruction needs to be relocated every time, frequent traps to the operating system prevents a reasonable performance. Vulcan creates a larger basic block and copies the instrumented code there if it cannot fit into the original place [21]. The head of the old basic block is replaced by a jump to the new one. Such a mechanism relocates the whole basic block, which has a large memory overhead. FIST does not relocate any instruction, but copies the replaced instructions (which may be incomplete) back to the original place for execution [13]. The extra copy-back action may cause a significant performance overhead.

Many binary instrumentation systems do instrumentation optimizations. Among them, probe coalescing is specially designed for probe-based instrumentation (first proposed by INS-OP [12]). We implement their algorithm in Dimension and get the similar result as their work.

Inlining has been adopted by many instrumentation systems to reduce the performance overhead. Several systems employ partial inlining that inlines the frequently executed part while leaving the other part as a normal analysis routine. INS-OP can use profiling information to employ partial inlining and Pin provides an API for a user to specify which part of the analysis routine she wants to inline [12, 15].

9. CONCLUSION

As interest in translation-based VEEs has been growing, it is important to build a wide variety of customized program analysis tools for both the source and target binaries that can serve diverse VEEs. We develop Dimension, an instrumentation tool that is specially designed for use in VEEs. The design of Dimension identifies the few components interfacing with VEEs and develops simple communication interfaces between the VEE and Dimension. With the two case studies, we demonstrate the flexibility of using Dimension. From experimental results, we show the effectiveness of optimization techniques applied in Dimension, the negligible effect introduced by flexibility of Dimension on performance, and the reasonable slowdown caused by Dimension compared with other instrumentation systems. We conclude that Dimension is a well-designed flexible, comprehensive, easy-to-use, and efficient instrumentation tool for programs in VEEs. Moreover, its use domain can easily extend to traditional execution environments.

ACKNOWLEDGEMENTS

This paper benefited from fruitful discussions with Naveen Kumar and Jonathan Misurda. We also thank the anonymous reviewers for their useful suggestions and comments on how to improve the paper.

REFERENCES

[1] M. Arnold, S. Fink, D. Grove, M. Hind and P. Sweeney. Adaptive Optimization in the Jalapeno JVM. In *Proc. of Conf. on Object-Oriented Programming Systems, Languages and Applications*, October 2000.

[2] V. Bala, E. Duesterwald and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of Conf. on Programming Language Design and Implementation*, June 2000.

[3] D. Bruening, T. Garnett and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of Intl. Symp. on Code Generation and Optimization*, March 2003.

[4] B. Buck and J. Hollingsworth. An API for Running Code Patching. *Journal of High Performance Computing Applications*, 14 (4), 2000.

[5] B. Cantrill, M. Shapiro and A. Leventhal. Dynamic Instrumentation of Production Systems. In *Proc. of Symp. on Operating Systems Design and Implementation*, December 2004.

[6] K. Hazelwood and D. Brooks. Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Optimization. In *Proc. of Intl. Symp. on Low Power Electronics and Design*, August 2004.

[7] Clara Jaramillo, Rajiv Gupta, Mary Lou Soffa. Capturing the Effects of Code Improving Transformations. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, October 1998.

[8] Clara Jaramillo, Rajiv Gupta, Mary Lou Soffa. Debugging and Testing Optimizers through Comparison Checking. *Electronic Notes in Theoretical Computation Science*, 65(2), 2002.

[9] The Java Technology website. http://java.sun.com/.

[10] The JRat website. http://jrat.sourceforge.net/.

[11] P. Kessler. Fast Breakpoints: Design and Implementation. In *Proc. of Conf. on Programming Language Design and Implementation*, June 1990.

[12] N. Kumar, B. Childers and M. L. Soffa. Low Overhead Program Monitoring and Profiling. In *Proc. of Workshop on Program Analysis for Software Tools and Engineering*, September 2005.

[13] N. Kumar, J. Misurda, B. Childers and M. L. Soffa. Instrumentation in Software Dynamic Translators for Self-Managed Systems. In *Proc. of Workshop of Self-Managing Systems*, November 2004.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of Intl. Symp. on Code Generation and Optimization*, March 2004.

[15] C. Luk. R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of Conf. on Programming Language Design and Implementation*, June 2005.

[16] J. Maebe, M. Ronsse and K. De Bosschere. Diota: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Comp. of Workshops and Tutorials held in conjunction with the 11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, September 2002.

[17] J. Misurda, J. Clause, J. Reed, B. Childers and M. Soffa. Demand-driven Structural Testing with Dynamic Instrumentation. In *Proc. of Intl. Conf. on Software Engineering*, May 2005.

[18] N. Nethercote. *Dynamic Binary Analysis and Instrumentation or Building Tools is Easy*. Ph.D. Thesis, University of Cambridge, November 2004.

[19] K. Scott and J. Davidson. Safe Virtual Execution Using Software Dynamic Translation. In *Proc. of the 18th Annual Computer Security Applications Conf.*, July 2002.

[20] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson and M. L. Soffa. Reconfigurable and Retargetable Software Dynamic Translation. In *Proc. of Intl. Symp. on Code Generation and Optimization*, March 2003.

[21] A. Srivastava, A. Edwards and H. Vo. Vulcan: Binary Transformation in A Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[22] A. Srivastava and A. Eustace. Atom: A System for Building Customized Program Analysis Tools. In *Proc. of Conf. on Programming Language Design and Implementation*, June 1994.

[23] The Transitive website. http://www.transitive.com/.