DrPin: A dynamic binary instumentator for multiple processor architectures

Luis Fernando Antonioli[1], **Ricardo Pannain**[1], **Rodolfo Azevedo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – Cidade Universitária – Campinas – SP – Brazil

luis.antonioli@students.ic.unicamp.br, {pannain,rodolfo}@ic.unicamp.br

Abstract. *Modern applications rely heavily on dynamically loaded shared libraries, making static analysis tools used to debug and understand applications no longer sufficient. As a consequence, dynamic analysis tools are being adopted and integrated into the development and study of modern applications. Building tools that manipulate and instrument binary code at runtime is difficult and error-prone. Because of that, Dynamic Binary Instrumentation (DBI) frameworks have become increasingly popular. Those frameworks provide means of building dynamic binary analysis tools with low effort. Among them, Pin 2 has been by far the most popular and easy to use one. However, since the release of the Linux Kernel 4 series, it became unsupported, and Pin 3 broke backward compatibility. In this work we focus on studying the challenges faced when building a new DBI (DrPin) that seeks to be compatible with Pin 2 API, without the restrictions of Pin 3, that also runs multiple architectures (x86-64, x86, Arm, Aarch64), and on modern Linux systems.*

1. Introduction

Throughout the history of modern computing, programs' complexity has been growing at a fast pace and the need for tools that help users understand program behaviors are on the rise. Modern applications and languages frequently rely on dynamically linked shared libraries, dynamic code generation, and other features that are only defined at runtime. The only viable option to grasp a precise understanding of these programs is to analyze them at runtime. Among the many research fields that attempt to tackle these problems, code instrumentation and analysis is one of the most popular ones. Dynamic instrumentation of binaries is one of the most powerful type of instrumentation because it does not require users to hold the source code of the application, nor it requires recompilation or the rewrite of the binary.

Because of these advantages, the number of DBI (Dynamic Binary Instrumentation) frameworks and tools based on them has grown. However, each framework has its own API (application program interface) and often we are left with tools that only work in some architectures or operating systems, due to inherited limitations from the chosen DBI framework, instead of a limitation in the tool itself.

Among the existing dynamic binary instrumentation frameworks, DynamoRIO [Bruening et al. 2003], Valgrind [Nethercote and Seward 2007], and Pin [Luk et al. 2005a] are widely used as they provide APIs that facilitate the creation of a wide spectrum of tools. From tools that detect memory allocation

errors [Seward and Nethercote 2005] and do cache simulation [Nethercote 2004], to tools that analyse malicious programs [Sinnadurai et al. 2008].

Several studies in the field of program instrumentation are available in the literature and some will be described in section 2, where they will be better categorized. Among these works, Nethercote et al. stands out in [Nethercote and Seward 2007] describing how Valgrind differentiates itself from other DBIs, showing the features that allow Valgrind to support complex analysis tools while its intermediate representation allows it to work with multiple architectures and operating systems. Moreover, several publications [Carlson et al. 2011, Sanchez and Kozyrakis 2013, Mutlu and Moscibroda 2007, Miller et al. 2010] show that there are a large number of tools that have been developed based on the Pin framework. Lastly, in [Bruening and Amarasinghe 2004], Bruening et al. details into a great length the challenges faced by the authors to design and implement DynamoRIO: a DBI framework focused on transparency that is capable of running heavily multi-threaded applications and large scale real-world applications.

This work aims to study and implement a new DBI, called DrPin, which is built on top of the DynamoRIO infrastructure and is compatible with the Pin 2 API. DrPin's goal is to bring together the easy-of-use of Pin 2, without the restrictions of Pin 3 and merge them with the support of multiple architectures (x86-64, x86, Arm, Aarch64) from DynamoRIO, being able to run on modern Linux systems.

This paper is organized as following: Section 2 presents the related work, Section 3 presents DrPin and its internals while Section 4 shows the support of multiple platforms and experiments. Finally, Section 5 presents our conclusions.

2. Related Work

The world of DBI frameworks is rich. Some of them are focused on specific environments [Villa et al. 2019, Ravnås 2016], while others serve a broad spectrum of applications [Nethercote and Seward 2007, Luk et al. 2005a, Bruening et al. 2003]. From the userbase perspective, three of them stand out: Valgrind, DynamoRIO, and Pin. Therefore, in this section, we highlight their primary features and differences.

2.1. Valgrind

Valgrind [Nethercote and Seward 2007] is a dynamic binary instrumentation (DBI) framework focused primarily on complex and heavy analysis. To be able to do so, it has a robust infrastructure containing intermediate code representation and shadow values for registers and memory. It has its code entirely available under the GNU General Public License (GPL), supports multiple architectures (PPC, PPC64, ARM, ARM64, x86, AMD64, MIPS32, MIPS64, S390X), and has popular tools such as Memcheck.

A key difference between Valgrind and the others DBI frameworks is its intermediate language representation before applying instrumentation instructions, which allows the users to work with a very wide range of architectures at the cost of some performance. For instance, the Memcheck tool had a slowdown of 22 on average on the SPEC CPU2000 benchmark [Nethercote and Seward 2007].

Valgrind's architecture is divided into two parts: the core and the tools (plugins). The core is responsible for providing the infrastructure to support instrumentation. Therefore it provides many services, such as a JIT compiler, memory management, threads

scheduler, and an error recording system. The tools, however, are responsible for defining how the program should be instrumented. The purpose of this architecture is to make the implementation of the tools as light as possible.

Because Valgrind operates in user space, it is unable to instrument or translate the execution of instructions that occurs within system calls, since it does not have access to kernel code.

2.2. DynamoRIO

DynamoRIO [Bruening et al. 2003, Bruening and Amarasinghe 2004] is a runtime code manipulation system that supports code transformations anywhere in the program during its execution. It has an API that allows its users to develop dynamic tools for a wide variety of functions.

One feature that sets DynamoRIO apart from other DBI's is that it does not just insert analysis and instrumentation instructions into the middle of the application program, but also allows the application instructions themselves to be manipulated. This is because, when designed, its primary goal was to be a runtime code manipulation system, and since it is a runtime code manipulation system, it can instrument applications as well. It has support for IA-32, AMD64, ARM, and AArch64.

2.3. Pin

Pin [Luk et al. 2005b] is a dynamic binary instrumentation (DBI) framework for IA-32, x86-64, and MIC architectures that enables the creation of dynamic analysis tools. One of the strengths of Pin today is the number of projects and tools [Soares et al. 2018] built using it as the infrastructure, making it the engine of many simulators, emulators, as well as being used for the study of architectures. Sniper [Carlson et al. 2011] and ZSim [Sanchez and Kozyrakis 2013] are examples of two major projects that use Pin as infrastructure. Like DynamoRIO and Valgrind, Pin has its instrumentation based on a JIT compiler that translates the application code to the instrumented code at runtime.

As with Valgrind and DynamoRIO, it also has a division between the tool code (what Pin calls a "pintool") that is used to define how the application should be instrumented, and the rest of the infrastructure (Pin). All the interaction of the writer of a new tool (pintool) and the Pin infrastructure is done through an API, and just like Valgrind, all the Pin code, including infrastructure, is executed in user space, so that it is not possible to instrument code executed in the system kernel.

2.3.1. Pin evolution through time

Although DynamoRIO and Valgrind remained fairly backward-compatible during their existence, in 2016 Intel released Pin 3.0 with some big changes that affected its whole user base. During Pin 3.0 announcement, Intel decided that the Pin framework should use PinCRT C runtime. PinCRT was introduced into Pin framework to improve the portability of pintools across compilers and operating systems. By using PinCRT, pintools authors would be presented with a consistent behavior across many system interfaces and C/C++ routines, all of that across all supported operating systems.

2.3.2. PinCRT

PinCRT is defined as an OS-agnostic, compiler-agnostic runtime. It is composed of three layers of a generic interface that practically isolate all interactions between the pintools and the host operating system. The three layers are:

- A generic operating system API which provides functions like thread control and process control.
- A C runtime which provides standard C library implementation.
- A C++ runtime which does the same, but for the C++ standard library.

It's worth noting that several restrictions come along with PinCRT. Tools cannot make any system calls or link with any system libraries, and are obligated to use PinCRT instead of any system runtime. The C++ runtime also does not support C++11 and RTTI (Run-Time Type information). For this reason, the famous C++ Boost library [Karlsson 2005] cannot be used.

2.3.3. Pin 2 deprecation

The future of the Pin framework is along with PinCRT. Since the last release of Pin 2 was made in February 2015, PinCRT is here to stay. Unfortunately, a great number of pintools already written uses Pin 2. The more complex they are (like the pintool that is the heart of the ZSim simulator) the harder it is for them to migrate to Pin 3. This is because, to use PinCRT, which is necessary in order to use Pin 3, the pintool, as well as all its dependencies, need to comply with PinCRT deficiencies.

To be able to upgrade to Pin 3, pintools that have dependencies need to drop all dependencies that use native system libraries (ex: libc, pthreads, etc) or rewrite them, replacing those native system libraries with PinCRT counter-parts.

Because of the reasons listed above, many pintools have not yet been ported to Pin 3. Staying with Pin 2 also brings many challenges, namely:

- Pin 2 only supports the Linux kernel 3.* series.
- Only GCC up to version 4.* is supported.

Because Pin is a closed-source software under Intel property, the community around it cannot provide support for it, proposing patches that possibly would make their pintools continue to work in modern Linux distributions.

Table 1 summarizes the main differences between the presented DBI frameworks, also including DrPin.

3. The DrPin Dynamic Binary Instrumentation Framework

Considering the problems shown in section 2.3.3, since the deprecation of Pin 2, a gap in the DBI framework spectrum appeared. In this work, we present DrPin, a DBI framework to address this gap.

DrPin is an open-source dynamic binary instrumentation framework that aims to have its API compatible with Pin 2, at the same time that supports multiple architectures (aarch64, arm, x86, and amd64) while also running on modern Linux kernel series.

Table 1. DBIs summary

	Pin 2	Pin 3	DynamoRIO	Valgrind	DrPin
API easy to use	Yes	Yes	No	No	Yes
Supports Linux kernel >= 4.x	No	Yes	Yes	Yes	Yes
Supports GCC >4.x	No	Yes	Yes	Yes	Yes
Supports x86/x86-64	Yes	Yes	Yes	Yes	Yes
Supports Arm/Aarch64	No[1]	No	Yes	Yes	Yes
Speed	Fast	Fast	Fast	Slow	Fast
Tool can use system libraries	Yes	No[2]	Yes	Yes	Yes

[1]Dropped support on winter 2012
[2]Tools can only use PinCRT for interfacing with the hosting system

We decided to make our DBI framework on top of another well-established open-source framework (DynamoRIO) primarily because making it from scratch would be a very extensive and laborious task. Moreover, it would deflect efforts from our focus and interest, which is to be able to provide great compatibility with the Pin 2 application programming interface so that many of the tools that were written using Pin 2, and since the release of Linux Kernel 4.0 are no longer supported, can be used again. As Table 1 summarized, DrPin combined the positive aspects of DynamoRIO with the easy-of-use of Pin 2, at the same time that it did not incorporate the restrictions that Pin 3 brought to pintool writers.

3.1. DynamoRIO as baseline

While studying DBI frameworks, it was noticeable that in this arena, besides Pin, two frameworks were dominants: DynamoRIO and Valgrind. Both are stable and have organically created a community around them. In popularity, Valgrind takes the lead as it powers one of the most popular dynamic analysis tools used by many C/C++ developers, which is called memcheck.

Examining Valgrind closely we observed that its core is very different from Pin and so does its API. Because Valgrind aims to support a vast amount of architectures, while also making the process of supporting new architectures as easy as possible, its authors decided to architecture it around a disassemble-and-resynthesize (D&R) paradigm. Valgrind does not instrument the binary code directly. Instead, it first translates the application code into an IR (intermediate representation), transforming each application instruction into one or more IR operations. Then it instruments the application by adding additional IR operations into the program. Finally, it compiles the intermediate representation back to binary code and then executes it. This process makes Valgrind slower than both Pin and DynamoRIO.

DynamoRIO, on the other hand, implements a JIT compiler that operates directly into the application binary, modifying application instructions and inserting instrumentation instructions as the application runs, just like Pin does, without using an intermediate representation. As consequence, the DynamoRIO provides an API that is closer to Pin.

3.2. Architecture

DrPin is built as a DynamoRIO client (which is analogous to a pintool in the Pin world). We decided to architecture it this way, instead of modifying DynamoRIO core directly, to avoid

the need to maintain a fork of DynamoRIO, patching every change made on the official DynamoRIO code repository. This decision restricted DrPin to use only DynamoRIO's public interface, meaning we cannot take advantage of any private information that is only available on DynamoRIO's core. Notice that Pin also implements this policy.

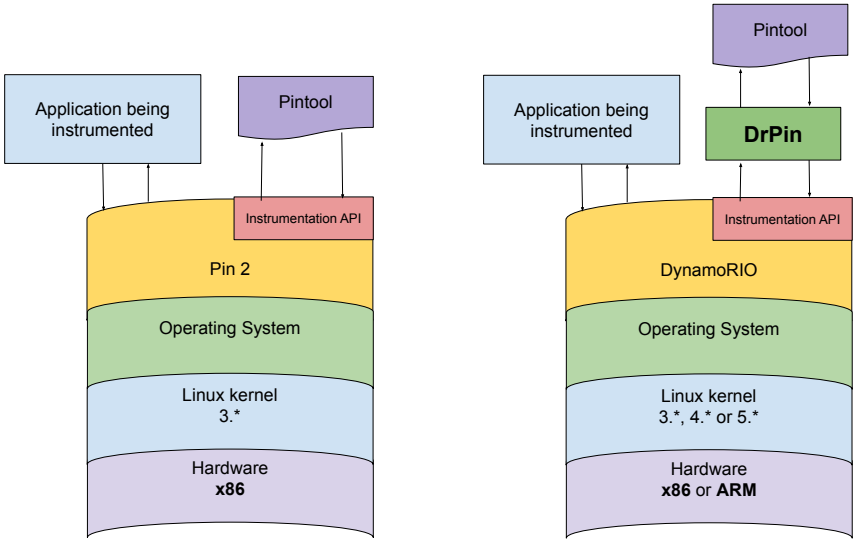

Figure 1. DrPin Overview

Figure 1 highlights the key points that distinguish DrPin from the original Pin 2 framework. While Pin 2 is only capable of running on old Linux kernels (3.* series) and on the x86/x86-64 platform, DrPin is capable of running in modern Linux kernels (4.* and 5.* series) while also running on Arm/Aarch64 and x86/x86-64. DrPin also supports all available gcc versions. As represented in the figure, DrPin itself only interacts with DynamoRIO through its public interface and never interacts directly with the application being instrumented. All the interactions between DrPin and the instrumented application are done through DynamoRIO. On the other hand, DrPin is responsible for interacting with the pintool provided by the user, thus receiving API calls from the pintool and finding ways to fulfill them using only DynamoRIO API functions.

Because there is a disparity in the information provided by both Pin's and DynamoRIO's API (not only in terms of granularity, but also in format), there is no simple way of mapping every single Pin API call into one or more DynamoRIO API functions. Instead, DrPin approaches this problem by gathering as much information as it can from the DynamoRIO engine, constructing a light-weight model of the running application, and then using this model to fulfill the requests made by the pintool. Another factor that played an important role in our decision of keeping such light-weigth model (despite the potential of increasing DrPin overhead) is the fact that DynamoRIO's API does not provide all the application's life-cycle events specified in Pin's API. Our approach allows not only DrPin to synthetically create events related to the life cycle of the instrumented application that DynamoRIO's API lacks and were found in Pin's API, but also allows DrPin to complement the events emitted by DynamoRIO with additional information

needed to match Pin API specifications.

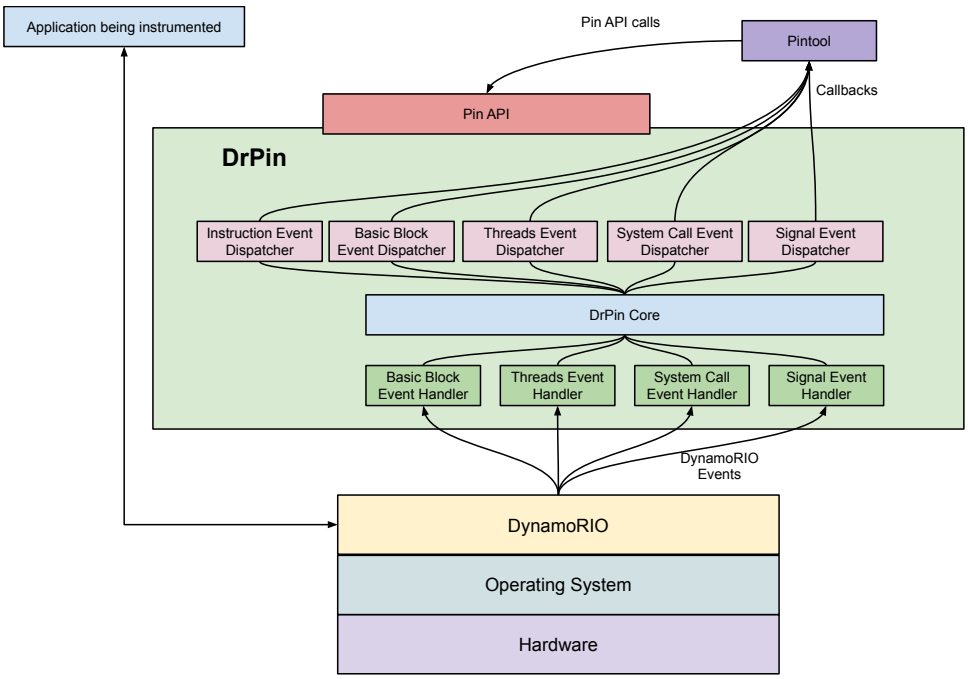

Figure 2. DrPin: Architecture

As represented in Figure 2, DrPin subscribes to all events DynamoRIO issues through its API. Doing so, DrPin creates and maintains internal data structures that help DrPin understand the state of the running application at any point in time. This information is also used by DrPin dispatchers to decide when to issue Pin events to the user-provided pintool.

4. Evaluation on multiple platforms

One of the aspects that sets DrPin apart from the original Pin 2 framework is the possibility of instrumenting not only x86/x86-64 applications, but also Arm/AArch64 ones. As expected, not all Pin API functions are suitable for the Arm/AArch64 platform, as some API functions are very specific to the x86/x86-64 platform (like `INS_IsRDTSC` which returns if an instruction is rdtsc or rdtscp), but supporting at least the more generic ones on both platforms makes the construction of pintools independent of platforms possible. One example of such platform-independent pintool is presented in Listing 1. The pintool collects the mnemonic of every instruction executed and at the end of the execution, displays a histogram of the 10 most executed mnemonics.

Although the pintool presented in Listing 1 needs to know the mnemonic of all executed instructions, it can rely on the `INS_Disassemble` API function (which works on all platforms) and do a simple `string` operation to extract the desired information. Doing so, the pintool can operate seamlessly on all platforms supported by DrPin.

```
typedef std::pair<std::string, int> mnemonic_counter;
std::vector<mnemonic_counter> mnemonic_histogram;

int mnemonicIdx(std::string &mnemonic) {
    {...}
}

VOID incrementMnemonicCount(int idx) {
  mnemonic_histogram[idx].second++;
}

VOID instrumentInstruction(INS ins, VOID *v) {
  std::string ins_disassembled = INS_Disassemble(ins);
  std::string mnemonic = ins_disassembled.substr(0, ins_disassembled.find(' '));
  int mnemonicIndex = mnemonicIdx(mnemonic);
  if (mnemonicIndex == -1) {
    mnemonicIndex = mnemonic_histogram.size();
    mnemonic_histogram.push_back(std::make_pair(mnemonic, 0));
  }
  INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)incrementMnemonicCount, IARG_UINT32,
    mnemonicIndex, IARG_END);
}

VOID fini(INT32 code, VOID *v) {
    sort_mnemonic_vector();
    print_top_10_mnemonics();
}

int main(int argc, char *argv[]) {
  if (PIN_Init(argc, argv)) {
    return 1;
  }
  INS_AddInstrumentFunction(instrumentInstruction, 0);
  PIN_AddFiniFunction(fini, 0);
  PIN_StartProgram();
  return 0;
}
```

Listing 1. Pintool: Top 10 mnemonics executed

The pintool 1 is very concise. It maintains an array of counters (one per mnemonic) and updates it before every execution of an instruction. To exemplify the functionality of such pintool, we instrumented the execution of the ubiquitous GNU `ls` program, present in almost all GNU/Linux distributions, during its operation of listing the contents of the `/etc` folder. Listings 2 and 3 shows the output of the pintool on the x86 system and on the AArch64 one respectively.

DrPin was able to execute the same pintool on two different systems. The first is an x86 system that runs Manjaro Linux with Linux Kernel `5.6.16-1-MANJARO`. The second is an Aarch64 system running Debian GNU/Linux 9 (stretch) with Linux Kernel `4.9.0-4-arm64` on a QEMU emulator 5.0.0.

```
  mov       : 150408
  cmp       : 47419
  add       : 44746
  jz        : 41268
  test      : 35868
  jnz       : 28868
  push      : 23115
  lea       : 22110
  pop       : 20314
  movzx     : 20208
```

Listing 2. Top 10 mnemonics: x86-64 output

```
  add       : 162417
  ldr       : 152696
  subs      : 134900
  orr       : 117405
  ldrb      : 67643
  str       : 64406
  stp       : 49024
  cbz       : 48960
  ldp       : 44350
  cbnz      : 38602
```

Listing 3. Top 10 mnemonics: AArch64 output

Although DrPin makes it easy for its users to write architecture-independent pin-

tools, it is possible that a pintool that only uses architecture-independent API function still does not run correctly on all platforms. This might happen because there are architecture details that may interfere with the correct operation of the pintool. To illustrate this issue, consider the Load-Exclusive and Store-Exclusive synchronization primitive present in the arm architecture. A pintool that happens to executes code between a linked load and store instruction might disrupt the application normal execution on arm while running fine on x86. Therefore, knowledge of the target architecture is still a valuable asset for the pintool writer.

4.1. Performance Optimizations

The Pin API is expressive and user-friendly, therefore it provides many facilities for the pintool writer that manifest itself in terms of rich API functions that provide many conveniences. Because DrPin does not operate inside DynamoRIO, many of these conveniences provided by the Pin API needed to be simulated by calling multiple DynamoRIO API functions, increasing DrPin overhead on those functionalities. This issue became most apparent while implementing `INS_InsertCall(INS ins, IPOINT action, AFUNPTR funptr, ...)` API function. This function is variadic and as the last parameter, it receives zero or more arguments that will be passed to `funcptr` as arguments at runtime. There is a wide variety of values that can be specified, from constants to processor architectural state at the time of the invocation, such as register values.

Pintools usually rely heavily on the `InsertCall` family of API functions (`BBL_InsertCall`, `RTN_InsertCall` and etc.) to obtain runtime information, and not supporting them would imply in a serious limitation for DrPin. The closest API function that DynamoRIO has to perform the same operation is `dr_insert_clean_call`. But DynamoRIO's function lacks a major functionality that makes the Pin variant especial: the ability to pass runtime information as argument to the analysis function. To work around this, DynamoRIO users normally request the needed runtime information inside the analysis function using other API functions.

To implement the `InsertCall` family of functions, whenever DrPin received a request to insert a call to an analysis function, DrPin registered a call to a helper function that collected the information and later called the analysis function with the necessary information.

This approach, however, implied that all the burden of parsing the function parameters and requesting the desired information from the DBI engine happened during the execution of the instrumented application. This caused an overhead even for analysis functions that did not need any runtime information, such as tracers that only operate on information that does not change in the course of the execution of the application (such as instruction addresses, function names and etc). To mitigate this problem, we optimized DrPin to inspect the requested function arguments and when it detects that no argument needs runtime information, DrPin registers the analysis function call using DynamoRIO's `dr_insert_clean_call` directly. This little optimization enabled DrPin to operate tracer pintools without almost no overhead when compared to DynamoRIO, and also preserved Pin valuable functionalities.

4.2. Performance Comparison

DBI frameworks have always paid additional attention to their performance because usually, DBI tools interact with every single instruction executed by the application. It is true,

however, that not all DBI tools are made equal. Some require little runtime information, being as simple as generating an instruction trace, while others do heavy analysis, creating and maintaining heavy-weight models to simulate how an application interacts with a hypothetical machine's hardware.

To better illustrate the performance characteristics of DrPin, we focused our attention on two distinct scenarios found in the DBI world. In the first scenario, we want to compare the slowdown of the DBI frameworks during the execution of lightweight analysis. To do so, we chose the instruction counter pintool, which has a very light analysis function (only increments a counter for every executed instruction) but instruments every single instruction, therefore can show the overhead of instrumenting all the instructions in an application.

Since Pin 2 cannot run on newer operating systems, we had to perform the tests on a Ubuntu 14.04 machine with Linux kernel 3.19. We used different compiler versions for both Pin and DrPin because GCC 4.8.5 did not fully support all the C++11 features used in DynamoRIO and DrPin codebase. We used the following specifications during the tests:

- **OS**: Ubuntu 14.04.6 LTS
- **Kernel**: 3.19.0-25-generic
- **Compiler**: GCC 7.4.0 for DrPin and GCC 4.8.5 for Pin
- **Processor**: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
- **RAM memory**: DDR4 16GiB @ 3200 MHz

We chose SPECint 2006 as our benchmark because Pin 2 could not run properly the SPEC CPU 2017 benchmark. After executing the whole SPECint 2006 benchmark, we saw that in this scenario, the performance of both DBI frameworks was close, with Pin 2 being a little faster than DrPin. On average, DrPin was 10% slower than Pin 2 and 11.6 times slower than the native execution.

To give a perspective of how the performance of both DrPin and Pin 2 compares with other DBI frameworks, we built Figure 3 that compares the performance of Pin 2, Pin 3, DrPin, and DynamoRIO frameworks while running the instruction count analysis for the first three programs of the SPECint 2006 benchmark. All DBI frameworks showed a similar performance during the execution of this lightweight analysis, with DrPin presenting an average result 9% and 9.8% worse than Pin 3 and DynamoRIO respectively.

In section 4.1 we explained that DynamoRIO does not provide a built-in way of implementing the `INS_InsertCall` function if runtime information is requested by the analysis function. Because of this, when an analysis function that needs runtime information is registered with `INS_InsertCall` API function, DrPin cannot insert a direct call the analysis function into the instrumented binary because it does not know the value of the arguments yet. Instead, it inserts a call to a function that, at runtime, will gather the necessary data and will later call the analysis function with the right arguments.

Therefore, in the second scenario, we designed an experiment to abuse this DrPin's weakness. We create a pintool that for every branch in the application, registers an analysis function that receives as argument whether or not the executed branch has been taken.

We believe that this kind of pintool is near the worst-case scenario of DrPin compared to Pin, because of all API functions that were implemented in DrPin that

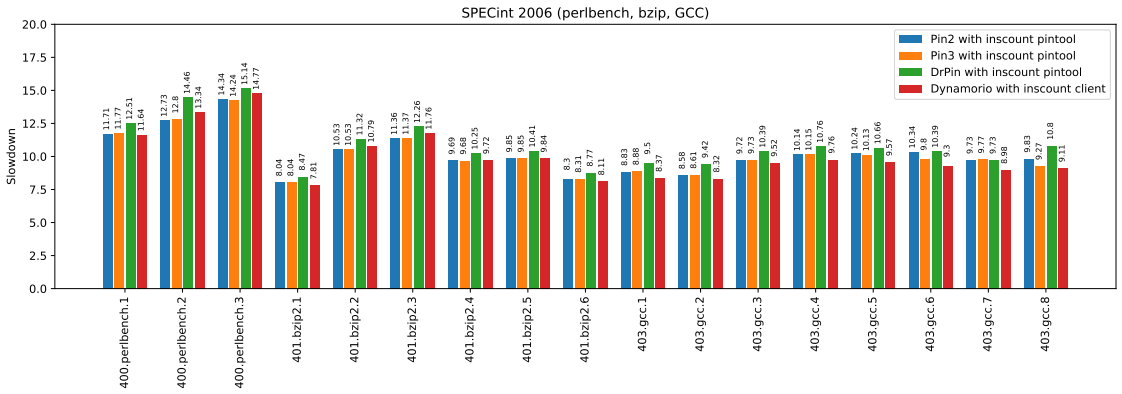

Figure 3. SPECint 2006 (perlbench, bzip, GCC): Instruction count execution slowdown

simulates a Pin API, this one has an overhead that is manifested during the execution phase of the instrumentation process and not only during the compilation phase. For this test, we used the same system specification described in the other performance tests, but instead of using the whole SPECint 2006 benchmark, we used only the `GCC` program with its eight different inputs. On average, Pin had a slowdown of 8.1 while DrPin had a slowdown of 499.8, therefore DrPin was two orders of magnitude slower than Pin for the giving scenario. When facing such slowdown, authors of pintool may want to use DynamoRIO APIs directly instead of using DrPin ones.

5. Conclusions

Our main objective in this work was to study and implement a new DBI, called DrPin, which was built on top of the DynamoRIO framework and is compatible with the Pin 2 API. DrPin's goal was to bring together the easy-of-use of Pin 2, without the restrictions of Pin 3. DrPin's supports multiple architectures (x86-64, x86, Arm, Aarch64) and runs on modern Linux systems, as it has DynamoRIO as its base. During the process, we studied and analyzed other existent DBI frameworks. We also encountered many challenges during the implementation of DrPin, since the differences in DynamoRIO and Pin API start to grow as we start implementing more complex API functions.

The source code of DrPin can be found in the following repository: `https://github.com/luisfernandoantonioli/drpin`. We hope it will be used in other future researches and by those who currently struggle with the deprecation of Pin 2. We encourage contributions and enhancements by the community.

6. Acknowledgments

This work is supported by the Sao Paulo Research Foundation (FAPESP) (2013/08293-7), CAPES (2013/08293-7), and CNPq (438445/2018-0, 309794/2017-0).

References

Bruening, D. and Amarasinghe, S. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

Bruening, D., Garnett, T., and Amarasinghe, S. (2003). An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE.

Carlson, T. E., Heirmant, W., and Eeckhout, L. (2011). Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE.

Karlsson, B. (2005). *Beyond the C++ standard library: an introduction to boost*. Pearson Education.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005a). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005b). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM.

Miller, J. E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. (2010). Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE.

Mutlu, O. and Moscibroda, T. (2007). Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, pages 146–160. IEEE Computer Society.

Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

Ravnås, O. A. V. (2016). Frida: A world-class dynamic instrumentation framework.

Sanchez, D. and Kozyrakis, C. (2013). Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486. ACM.

Seward, J. and Nethercote, N. (2005). Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30.

Sinnadurai, S., Zhao, Q., and fai Wong, W. (2008). Transparent runtime shadow stack: Protection against malicious return address modifications.

Soares, R., Antonioli, L., Francesquini, E., and Azevedo, R. (2018). Phase detection and analysis among multiple program inputs. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 155–161. IEEE.

Villa, O., Stephenson, M., Nellans, D., and Keckler, S. W. (2019). Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383.