Universidade Estadual de Campinas
Instituto de Computação

Luís Fernando Antonioli

DrPin: A dynamic binary instrumentator for multiple processor architectures

DrPin: Um instrumentador dinâmico de binários para múltiplas arquiteturas de processadores

CAMPINAS
2020

Luís Fernando Antonioli

DrPin: A dynamic binary instrumentator for multiple processor architectures

DrPin: Um instrumentador dinâmico de binários para múltiplas arquiteturas de processadores

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

Este exemplar corresponde à versão final da Dissertação defendida por Luís Fernando Antonioli e orientada pelo Prof. Dr. Rodolfo Jardim de Azevedo.

CAMPINAS
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

An88d

Antonioli, Luís Fernando, 1993-
DrPin : a dynamic binary instrumentator for multiple processor architectures / Luís Fernando Antonioli. – Campinas, SP : [s.n.], 2020.

Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de computação. 2. Códigos binários. 3. Análise dinâmica. I. Azevedo, Rodolfo Jardim de, 1974-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: DrPin : um instrumentador dinâmico de binários para múltiplas arquiteturas de processadores
Palavras-chave em inglês:
Computer systems
Binary codes
Dynamic analysis
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Fernando Magno Quintão Pereira
Guido Costa Souza de Araújo
Data de defesa: 16-10-2020
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5519-5090
- Currículo Lattes do autor: http://lattes.cnpq.br/5657260716636888

Universidade Estadual de Campinas
Instituto de Computação

Luís Fernando Antonioli

DrPin: A dynamic binary instrumentator for multiple processor architectures

DrPin: Um instrumentador dinâmico de binários para múltiplas arquiteturas de processadores

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP
- Prof. Dr. Fernando Magno Quintão Pereira
  DCC/UFMG
- Prof. Dr. Guido Costa Souza de Araújo
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 16 de outubro de 2020

Simplicity is prerequisite for reliability
(Edsger W. Dijkstra)

Agradecimentos

Gostaria de agradecer a todos que contribuiram direta ou indiretamente a este trabalho, em especial:

- Aos meus pais Antonio Sergio Antonioli e Lucia Helena Geraldo Antonioli, pela imensa dedicação, cuidado e apoio que foram fundamentais ao longo de minha vida.
- Ao meu irmão, Carlos Eduardo Antonioli, pela motivação, parceria e ensinamentos.
- Ao meu orientador, Rodolfo Azevedo, que com sua compreensão, sabedoria e experiência me guiou desde a graduação até o fim deste mestrado.
- Aos professores e funcionários do Instituto de computação da Unicamp, sempre dispostos a ajudar.
- Aos colegas do Laboratório de Sistemas de Computação (LSC), fundamentais para troca de conhecimento.
- Aos demais amigos que adiquiri durante graduação e o mestrado.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

O presente trabalho também foi realizado com apoio do CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (143197/2018-5).

Resumo

A complexidade dos programas está aumentando e as ferramentas usadas para seu desenvolvimento tem acompanhado tal evolução. Aplicações modernas dependem largamente de bibliotecas carregadas dinamicamente e algumas aplicações até geram código durante sua execução. Logo, ferramentas de análise estática, usadas para depurar e entender aplicações não são mais suficientes para se ter um panorama completo de uma aplicação. Como resultado, ferramentas de análise dinâmica (aquelas que são executadas durante o tempo de execução) estão sendo adotadas e integradas ao desenvolvimento e estudo de aplicacoes modernas. Entre essas, as ferramentas que operam diretamente no binário do programa são particularmente úteis no meio de inúmeras bibliotecas carregadas dinamicamente, onde o código-fonte pode não estar disponível. A construção de ferramentas que manipulam e instrumentam código binário durante sua execução é particularmente difícil e propensa a erros. Um pequeno erro pode resultar em um desvio completo do comportamento do programa sendo analisado. Por esse motivo, *frameworks* de Instrumentação dinâmica de binários (DBI) tornaram-se cada vez mais populares. Esses *frameworks* fornecem meios para criação de ferramentas de análise dinâmica de binarios com pouco esforço. Entre eles, o Pin 2 tem sido de longe o mais popular e fácil de usar. No entanto, desde o lançamento da série 4 do Linux Kernel, ele ficou sem suporte. Neste trabalho, nosso foco é voltado para o estudo dos desafios encontrados ao criar um novo DBI (DrPin) que tem como foco ser totalmente compatível com a API do Pin 2, ao mesmo tempo que também suporta várias arquiteturas (x86-64, x86, Arm, Aarch64) e sistemas Linux modernos. Atualmente, o DrPin suporta um total de 83 funções da API do Pin 2, o que o torna capaz de executar várias pintools originalmente escritas para o Pin 2 sem nenhuma modificação. Comparando o desempenho do DrPin com o Pin 2, para uma ferramenta simples que conta o número de instruções executadas, observamos que, para o benchmark `SPECint 2006`, somos, em média, apenas 10% mais lentos que o Pin e 11,6 vezes mais lentos que a execução nativa. Também exploramos um pouco o ecossistema em torno dos frameworks de instrumentação dinâmica de binários. Especificamente, estudamos e estendemos uma técnica que utiliza ferramentas de análise dinâmicas de binários, construida com a ajuda de frameworks DBI, para prever o desempenho de uma determinada arquitetura ao executar um programa ou benchmark específico, sem a necessidade de executar o programa ou benchmark inteiro. Em particular, estendemos a Metodologia SimPoint [31, 14, 15] para obter ganhos adicionais na redução do tempo necessário para obter tais previsões. Mostramos que, considerando as semelhanças no comportamento do programa entre diferentes entradas, podemos reduzir ainda mais o tempo necessário para obter resultados de simulação de benchmarks inteiros. Especificamente para `SPECint 2006`, mostramos que o número de SimPoints (diretamente proporcional ao tempo de simulação) pode ser reduzido em média 32%, perdendo apenas 0,06% da precisão quando comparado a técnica original. Diminuindo a precisão em 0,5%, observamos que o tempo de simulação é reduzido em média 66%.

Abstract

Programs' complexity is rising and the tools used in their development changed to keep up with this evolution. Modern applications rely heavily on dynamically loaded shared libraries and some of them even generate code at runtime, therefore static analysis tools used to debug and understand applications are no longer sufficient to understand the full picture of an application. As a consequence, dynamic analysis tools (those that are executed during runtime) are being adopted and integrated into the development and study of modern applications. Among those, tools that operate directly on the program binary are particularly useful in the sea of dynamically loaded libraries, where the source code might not be readily available. Building tools that manipulate and instrument binary code at runtime is particularly difficult and error-prone. A minor bug can result in a complete disruption in the behavior of the binary code being analyzed. 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. However, since the release of the Linux Kernel 4 series, it became unsupported. 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. In total, currently, DrPin supports a total of 83 Pin 2 API functions, which makes it capable of running many pintools originally written for the Pin 2 framework without any modification. Comparing the performance of DrPin to the original Pin 2 for a simple tool that counts the number of instructions executed, we observed that for the `SPECint 2006` benchmark we were, on average, only 10% slower than the Pin 2 framework and 11.6 times slower than the native execution. We also explored the ecosystem around DBI frameworks. Specifically we studied and extended one technique that makes use of dynamic binary tools, built with the help of DBI frameworks, to predict the performance of a given architecture when executing a particular program or benchmark without the need to run the entire program or benchmark. In particular, we extended the SimPoint Methodology [31, 14, 15] to obtain further gains in the time required to obtain the predictions. We showed that by taking into account similarities in the program behavior among different inputs, we can further reduce the time it takes to get simulation results of entire benchmarks. Specifically for `SPECint 2006`, we showed that the number of SimPoints (which is directly proportional to the simulation time) can be reduced by an average of 32% while losing only 0.06% of the accuracy when compared to the original technique. Further decreasing the accuracy by 0.5%, we observed the simulation time is reduced by an average of 66%.

List of Figures

List of Tables

Contents

Chapter 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.

Among the types of instrumentation, the dynamic instrumentation of binaries is one of the most powerful. This is because it does not require its 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 [3], Valgrind [26] and Pin [19] are widely used as they provide APIs that facilitate the creation of a wide spectrum of types of tools, from tools to detect memory allocation errors [33] and cache simulation [25] to the analysis of malicious programs [38].

Several studies in the field of program instrumentation are available in the literature and some will be described in chapter 2, where they will be better categorized. Among these works, Nethercote et al. stands out in [26] 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 [6, 32, 23, 22, 21] show that there are a large number of tools that have been developed based on the Pin framework. Lastly, in [2], 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.

1.1 Objective

Although Pin 2 is a very popular DBI and one of the easiest to use, it has several problems that we want to work around in this work. First, it has its source code closed, property of Intel Corporation, making its support and utility restricted to the company's interests.

Secondly, support for Pin 2 is restricted to Linux kernel series 3.x, causing great difficulties for its practicality in modern operating systems. Being outdated does not affect only the tools distributed directly with Pin, but also those that use it indirectly, as is the case of ZSim and the Sniper simulator. For purposes of illustration, the last Ubuntu LTS operating system that was released with support for the 3rd series of Linux kernel was released in April 2014.

Third, Pin 3, the successor of Pin 2, brought many new restrictions that made the transition almost impossible for several pintools. Namely, the pintool (and all its dependencies) cannot make any syscall or link to any system library. They are restricted to interact with the host operating system using only the PinCRT runtime.

Fourth, Pin currently only supports IA-32, x86-64 and MIC architectures, meaning that almost all IoT and mobile platforms are left outside of its reach.

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. In the context of DBI frameworks, we also want to explore and study the usage of DBI tools in other fields of computer science.

1.2 Contributions

The contributions made by this work can be divided into two parts. The first is DrPin itself. We designed, implemented and evaluated a new DBI framework capable of executing on multiple architectures and modern Linux operating systems. The second is the result of our study on practical applications of DBI based tools in other areas of computer science. As a joint effort with another graduate student from the Computer Systems Laboratory (LSC) at Unicamp, Rafael Mendonça Soares, we studied techniques to predict the performance of a given computer architecture when executing a particular workload based on information collected with the help of DBI tools.

More specifically, we extended the SimPoint methodology, which originally was intended to find redundancy on program behavior of a single program on a single input, to explore redundancy on program behavior for multiple inputs on the same program at once.

1.3 Structure

This work is structured as follows:

- Chapter 2 provides an overview of the basic concepts that surround our work. It also presents, in a condensed form, related work. We try to be brief and concise, but a reader familiar to the world of dynamic binary instrumentation frameworks might find the sections 2.1 and 2.2 skippable. However, section 2.3.3 is very important to the understanding of this work, so we recommend the reader to not skip it.
- Chapter 3 introduces DrPin, describes how it is architectured under the hood, explaining the reasoning behind it and compare with other existing DBI frameworks.
- Chapter 4 is the result of our exploration around the DBI framework ecosystem. Specifically, it is dedicated into presenting an extension of the SimPoint methodology, which leverages information about the program's behavior during its execution (obtained by the use of DBI tools) to predict the program's performance concerning many architecture metrics. However, the focus on this chapter is mainly on the SimPoint methodology and its extension.
- Chapter 5 provides an overall discussion about this work as well as some possible future work.
- Appendix A provides a list of all the Pin 2 API functions currently implemented by DrPin.

Chapter 2

Basic Concepts and Related Work

In this chapter, we go through the concepts and the related work that builds the basis for our work. We start the chapter describing the classes of analyses usually made upon a program. Next we go through three very popular DBI (*Dynamic Binary Instrumentation*) frameworks (Valgrind [26], DynamoRIO [3] and Pin [20]), describing their architectures, features and more importantly: their differences. Then, still in the topic of DBI frameworks, we present a little known DBI that seeks to be compatible with the Pin API called PEMU[42].

During our work we also explored some real use cases for DBI frameworks, therefore we continue the chapter elaborating about deterministic program replaying techniques, in particular, those who benefit from DBI frameworks to collect data about non-deterministic events to later recreate them in the same order they were collected during another execution of the program.

At last, we talk about techniques that focus on reducing the time spent on simulations used in computer architecture research. In special we talk about the SimPoints technique and how DBI frameworks and deterministic program execution tools that were built on top of DBI frameworks can be used to implement and validate it.

2.1 Program Analyses

With the increasing complexity of computer systems, tools that automatically extract program information have become increasingly prominent. A program analysis tool is generally analyzed from two perspectives: Whether the analysis is static or dynamic and whether the analysis is performed with the program binary only or if it uses its source code.

2.1.1 Static vs Dynamic Analysis

Static analysis involves analyzing program code (either binary or source) without executing it. This type of analysis is commonly used by many tools, compilers, and integrated development environments (IDEs), where they perform analyses to improve program optimization, remove unnecessary variable copies, improve memory management, and perform program correctness analysis. An interesting aspect of this type of analysis is that, since it does not execute the program being analyzed, its execution time is usually linked to the complexity and size of the program and not to the problem that the program is trying to solve or to an input in particular.

Dynamic analysis, on the other hand, involves analyzing the program as it executes. Some popular tools [32, 6, 33] perform dynamic program analysis. Among them, we can highlight profilers and debuggers. To perform such analysis, these tools need to insert their code between code snippets of the application being analyzed (either binary or source code) without the analysis code having any side effects on the application.

Both categories of analysis have strengths and weaknesses. While static analysis covers all possible (even the rarest) execution paths, it usually involves more complex algorithms and in some cases may be inaccurate as some program information can only be determined at runtime. Dynamic analysis, while not covering all execution paths, is typically simpler and more accurate.

2.1.2 Binary vs Source Code Analysis

In addition to the groups mentioned above, an analysis can also be characterized according to the type of code it works on.

Binary analysis is usually done in terms of simple structures such as instructions, memory addresses, and registers and can be performed on codes before the linker processing (object code) or codes after the linker processing (executable code).

As expected, source code analysis uses source code as the basis. Generally, the analysis is done on higher-level structures such as functions, expressions, and variables. This group also includes analyses that are made on top of structures that were extracted directly from the source code, such as control flow graphs.

In the same way as dynamic analysis and static analysis, binary and source analysis each have strengths and weaknesses. Source code analysis is often architecture-independent and sometimes even operating system independent. A weakness of this type of analysis is that it is specific to a language or language family, and requires the source code possession. By analyzing source code, this type of analysis can also gain access to high-level information in the application, allowing it to build more powerful and abstract tools.

Binary analysis, on the other hand, is generally architecture and operating system dependent, but it is independent of the programming language that was used to generate the binary. The biggest advantage of binary analysis is that it does not require the source code, so a much wider range of programs can be analyzed.

To summarize the characteristics of the types of analysis described above, we present Table 2.1 and Table 2.2

Table 2.1: Types of analysis: Static vs Dynamic Analysis

	Main aspect	**Advantage**	**Disadvantage**
Static Analysis	Does not execute the code being analysed.	Covers all possible execution paths.	Can be more inaccurate as some information are only available at runtime.
Dynamic Analysis	Executes the code being analysed.	Simpler and have all the information available at runtime, therefore can be more precise.	Only covers the execution paths that were taken during that particular execution.

Table 2.2: Types of analysis: Binary vs Source Code Analysis

	Main aspect	**Advantage**	**Disadvantage**
Binary Analysis	Built upon simple structures such as instructions, registers and memory addresses.	Is independent of the programming language and does not require the source code of the application therefore is applicable to a wider audience of programs.	Architecture and operating system dependent.
Source Code Analysis	Built upon higher level structures such as functions, expressions and variables.	Architecture independent and sometimes even operating system independent.	Specific to a language (or a family of languages) and requires the source code of the application.

2.2 Dynamic binary instrumentation vs Static binary instrumentation

Earlier in the chapter, we saw four ways in which an analysis can be categorized. Our work is focused on dynamic analysis, which is usually performed by inserting new instructions into the application code to collect information about its execution. The process of inserting instructions into the program code to gather information about its behavior is called *instrumentation*.

Performing dynamic analysis requires programs to be instrumented with analysis code. Thus, there are two ways to instrument a program: dynamic instrumentation and static instrumentation. Since during this work we are only interested in binary instrumentation, we will restrict our discussion to the differences between dynamic binary instrumentation and static binary instrumentation.

Static binary instrumentation rewrites the executable code or object code by entering the instrumentation code before it is executed. Dynamic binary instrumentation (DBI) occurs during program execution and instrumentation code is injected in the middle of program execution.

The biggest advantage of dynamic binary instrumentation, compared to static instrumentation, is that it does not require the program being instrumented to be preprocessed, and therefore, has no problem instrumenting code that is dynamically generated at runtime or has data mixed with the code.

However, dynamic binary instrumentation has disadvantages. Because the entire process is done during program execution, the impacts on execution performance are greater, and the entire instrumentation process is repeated with each new program execution.

2.3 Dynamic binary instrumentation frameworks

In this section, we present three popular DBI frameworks: Valgrind, DynamoRIO, and Pin. We briefly discuss their architectures, features, and differences. At the end of this section, we also present a little known DBI framework called PEMU, that seeks to be compatible with The Pin API.

2.3.1 Valgrind

Valgrind [26] 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 but cost you somewhat in performance.

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:

1. JIT compiler
2. Memory management
3. Signal handling
4. Threads Scheduler
5. 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.

However, although Valgrind does not insert instrumentation code into system calls, it must intercept all system calls of the application it is instrumenting to make sure that it does not lose control over the program being instrumented.

Signal handling, as with all DBIs, is a delicate part of the system. This is because when an application defines a signal handler, it is giving the kernel a memory address that is in the application space to call so that the kernel can notify it of a signal. If nothing was done by the DBI, the signal handler code would be called by the kernel directly, causing it to not be translated and instrumented by the DBI. Another bad scenario could also happen: If the signal handler does not return or use a *longjmp*[1], the DBI would permanently lose control over the application.

To work around this problem, Valgrind intercepts system calls that are used to register signal handlers and write down their addresses. Next, it registers with the kernel its own handlers so that if any signal is delivered by the kernel, it can pass on to the application without risk of losing control of the execution.

2.3.2 DynamoRIO

DynamoRIO [3, 2] 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,

[1]Longjmp is a C standard library function that restores the stack environment and execution location that was previously saved in a buffer by the setjmp C standard library function. Longjmp can be seen as a nonlocal `goto`.

when designed, its primary goal was to be a runtime code manipulation system, therefore it can instrument applications as well. It has support for IA-32, AMD64, ARM, and AArch64.

To ensure good performance, it modifies one basic block at a time and uses a cache to store previous translations.

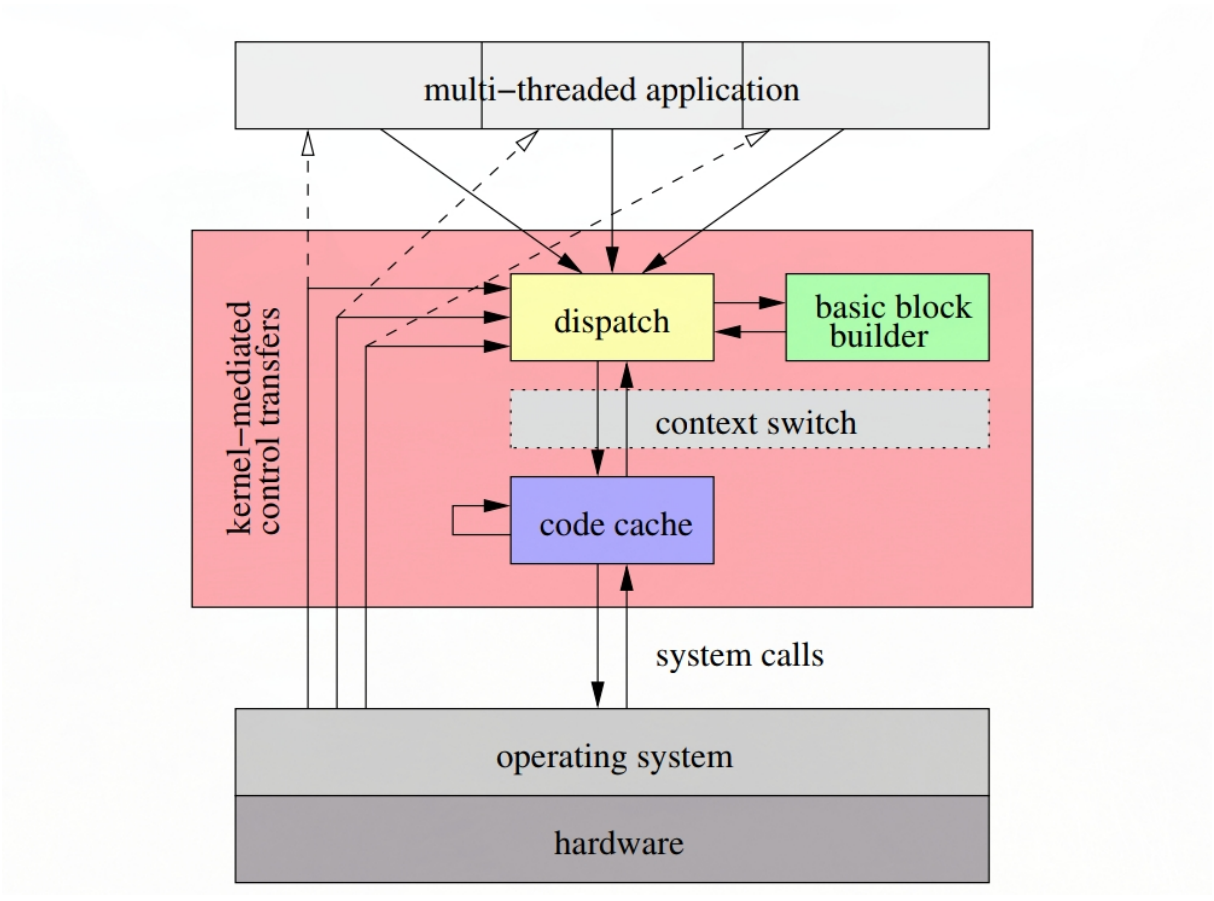


Figure 2.1: DynamoRIO's architecture. Image taken from [2]

In Figure 2.1 we show a simplified diagram of DynamoRIO architecture. DynamoRIO stands between the application and the operating system. Note that it executes only the code that is being generated and placed in its cache and has to intercept system calls to the kernel to ensure that it does not lose execution control on the way back of those calls.

DynamoRIO is capable of running multi-threaded desktop and server business applications with a performance penalty of typically thirty percent [2].

Transparency

A highly desired quality for a DBI is that it must be as transparent as possible. Being transparent, in this context is being invisible to the application being instrumented. This feature is not only desired by those who want to study the behavior of malicious applications but also for every user of the DBI. When analyzing applications, we want to see their behavior as similar as possible of when the instrumentation is not being applied.

Many applications have hidden bugs, i.e. accesses wrong addresses in the memory, that could change their behavior while running inside the DBI framework.

One source of conflict between the application that is being instrumented and the code that is instrumenting it are shared libraries. Many libraries have non-re-entrant routines. These routines could cause side effects from one code to another. To deal especially with this problem, DynamoRIO created a *private loader*. The private loader is responsible for loading all libraries that the instrumentation code needs into a memory address separated from where the libraries used by the application resides.

Following the example of shared libraries, DynamoRIO also makes a private stack for it to avoid conflicts with applications that access invalid positions in the stack or uses a hand-crafted code that uses the stack pointer as a general-purpose register.

Signals and System Calls

When dealing with signals, DynamoRIO registers its own handler instead of the one provided by the application by modifying all signal system calls. DynamoRIO uses only one handler for all signal types. After receiving a signal, its handler function simulates the same signal to the application handler. DynamoRIO also intercepts all system calls made by the application.

2.3.3 Pin

Pin [20] 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 is the number of projects and tools built using it as the infrastructure, making it the engine of many simulators, emulators as well as a tool for the study of architectures. Sniper [6] and ZSim [32] are examples of two major projects that use Pin as infrastructure.

Like Valgrind, Pin has its instrumentation based on a JIT compiler that translates the application code to the instrumented code at runtime.

The application's original behavior is not disturbed by Pin as it preserves both instruction and data addresses as well as values in registers and memory. This makes the information collected more relevant and accurate. To give an example of why this is important, some applications erroneously access data beyond the top of the stack and if Pin changes the application stack it would be changing the application behavior as well.

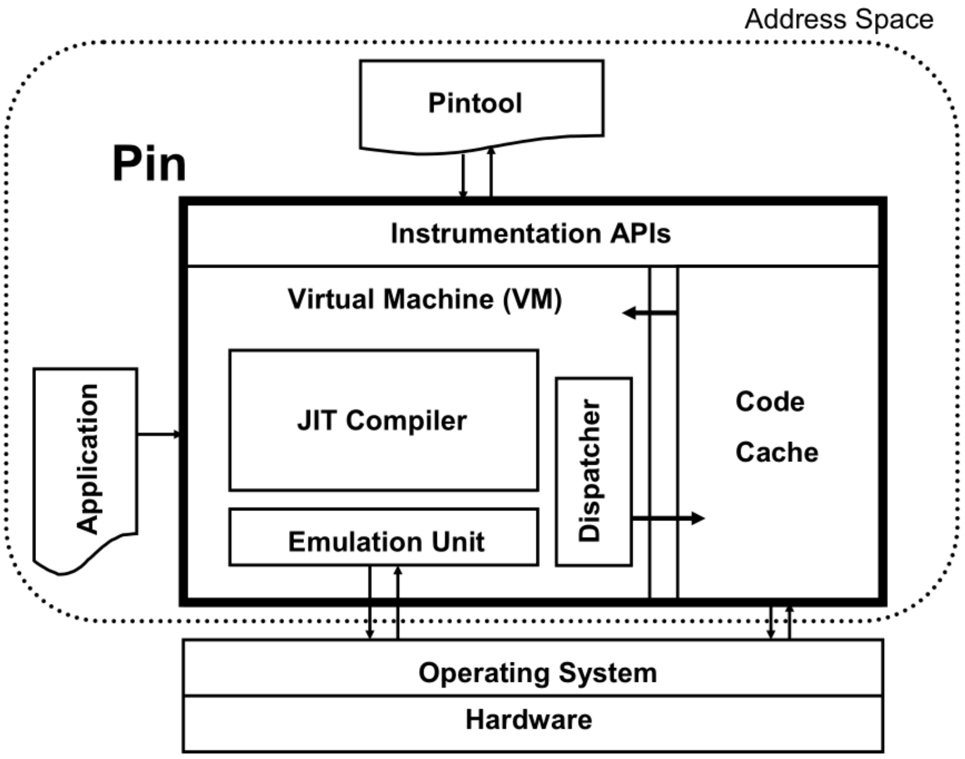


Figure 2.2: Pin's architecture. Image taken from [19]

In Figure 2.2 we have a small architecture diagram of Pin. As with Valgrind and DynamoRIO, it also has a division between the tool code (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.

Pin Injection

Pin loads itself into the address space of the application and uses the Unix Ptrace API to gain control over the application it is instrumenting. As soon as Pin is loaded, it loads the user pintool into the same address space of the application. Because Pin uses the Ptrace API as many debuggers do, it can attach to a running process and start jitting as well as leave the process. In this case, the application continues to run after Pin exit without any instrumentation.

JIT Compiler

Differently from Valgrind, Pin compiles one ISA directly into the same ISA, without any intermediate representation of code. The compiled code is stored into a code cache and the VM inside Pin (See figure 2.2) only executes code from the code cache. If necessary, Pin changes the address of branches to ensure it never loses control over the application that is being instrumented.

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 said that the Pin framework now uses 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.

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. The libc provided is based on the bionic libc, which is the libc used in the Android operating system.
- 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.
- The C++ runtime does not support C++11 and RTTI (Run-Time Type information). For this reason, the famous C++ Boost library [18] cannot be used.
- Tools are obligated to use PinCRT instead of any system runtime

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, it seems PinCRT is here to stay. Unfortunately, a great number of pintools already written uses Pin 2 and 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, as said earlier, 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 restrictions.

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

This process is problematic, because:

- Not everyone has the source code of all their dependencies to be able to recompile them.

- PinCRT only provides limited support for the vastly rich functionalities provided by native system libraries.

Because of the reasons listed above, many pintools have not yet been ported to Pin 3 series, and staying with Pin 2 brings many challenges, namely:

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

To illustrate how difficult is to work with the restrictions listed above, the last Ubuntu LTS release that follows these restrictions is Ubuntu 14.04 LTS (end of life 2019). As a result, it is very difficult to continue using all these tools in modern operating systems. Figure 2.3 summarizes the above discussed software compatibility problems and contrasts with the DynamoRIO situation.

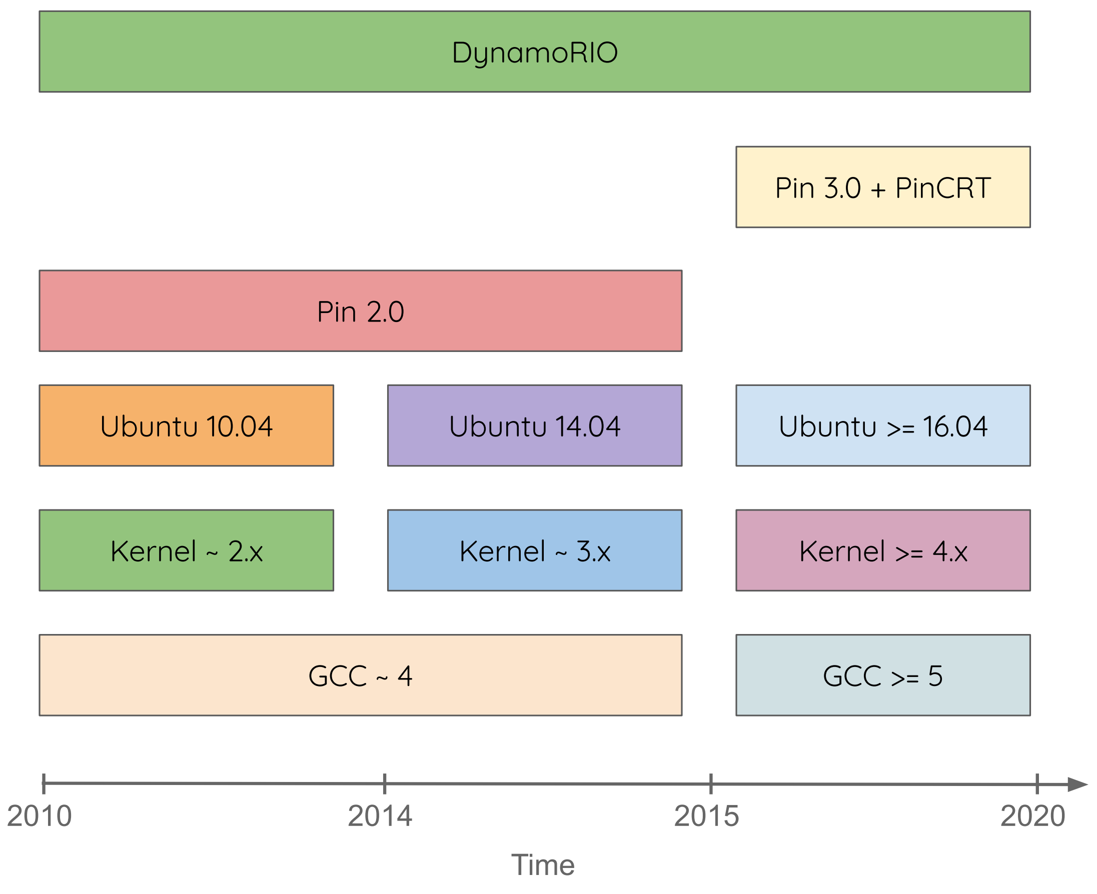


Figure 2.3: Pin vs DynamoRIO: software compatibility

Because Pin is a closed-source software under Intel property, the community around Pin cannot provide support for pin 2, proposing patches that possibly would make their pintools continue to work in modern Linux distributions.

2.3.4 PEMU

In 2015, Zeng *et al.* presented PEMU [42], an open-source DBI framework built on top of QEMU [1] targeting only the i386 architecture. PEMU major feature is being able to not only instrument user code, but also instrument the execution of kernel code, therefore performing a more comprehensive analysis of the system as a whole. To accomplish this, the authors used the QEMU emulator and did what they called *out-of-vm* instrumentation. By *out-of-vm*, the authors meant that the instrumentation lives below the operating system kernel. In the PEMU case, the framework is built as a software layer highly coupled with QEMU, sitting between QEMU and the guest operating system. In contrast, popular DBI frameworks like Valgrind, Pin and DynamoRIO share the same address space as the application they instrument. PEMU is not unique in its attempt to perform operating system level instrumentation, as others before it [41, 4] already explored this territory. Nevertheless, what distinguishes it from others is the attempt of the authors to provide the same instrumentation API as the popular Pin framework provides.

In the paper, the authors claim they implemented over one hundred Pin compatible APIs, but inspecting the project repository provided, we could only find sixty implemented APIs, being most of them related to Intel X86 encoder-decoder (Intel XED). Moreover, the rest of the implemented API is quite poor on functionality, usually lacking all functionality and sometimes being even completely empty. For instance, the arguably most important Pin API: *INS_InsertCall*, which inserts a call to an analysis function (the base block of the whole Pin framework), lacks almost all of the functionality that makes it useful. In particular, PEMU implementation of this API lacks the ability to obtain any runtime information like the branch target or whether a branch was taken or not.

Because of that and the fact that the project was not updated since its publication, we believe PEMU was primarily a proof of concept and was not meant to actually compete with other DBI frameworks.

2.4 Deterministic replay

As multi-core processors become increasingly popular and ubiquitous, complex and parallel applications are deployed everywhere. One of the many challenges of writing and debugging parallel programs is their non-repeatable behavior. The non-repeatable behavior could be caused by many sources: from the operating system that is providing resources for the application to the unsynchronized access to shared data (commonly known as *data race*) made by the application itself. Further, this unpredictable behavior makes debugging and analyzing complex parallel programs much more time consuming and unreliable.

There are a number of papers proposing methods to record the execution of a program, capturing all sorts of non-deterministic events, and later replaying the execution reproducing all those captured events now in a deterministic fashion [7] [34], but usually these approaches require special environments which can restrict its user base [7].

2.4.1 PinPlay

PinPlay [30] is a closed-source framework focused on capturing, deterministically replaying, and analysing the execution of large programs, while still being easy to use. The framework is based on the Pin DBI framework which only operates in user-space, not requiring any special environment to be used.

Under the hood, the PinPlay framework actually consists of two pintools: A logger, which is responsible for recording a program's execution and storing its information in a set of files collectively called *pinball* and a replayer, which runs off a pinball repeating the previously captured execution.

As may be noticed, just replaying the execution is of little use if it is not combined with a way to inspect the program being replayed.

Because PinPlay is built on top of Pin, it stands out its field by allowing the user to instrument program execution during the replay phase using the Pin framework equipped with a user-provided pintool.

How it works

PinPlay can be used to record both multi-thread and single-thread executions since even during single-thread execution, non-deterministic events occur and the record/replay infrastructure is useful. Nonetheless, under the hood PinPlay treats these two situations differently: For single-thread workloads, it replays the same sequence of instructions recorded during the logging phase following an absolute global order, but for multi-threaded workloads, it does not guarantee the same global ordering among instructions from different threads, except for the ones who interact with a shared memory space.

In Figure 2.4 we can see an abstracted overview of the components described earlier that together compose the PinPlay framework.

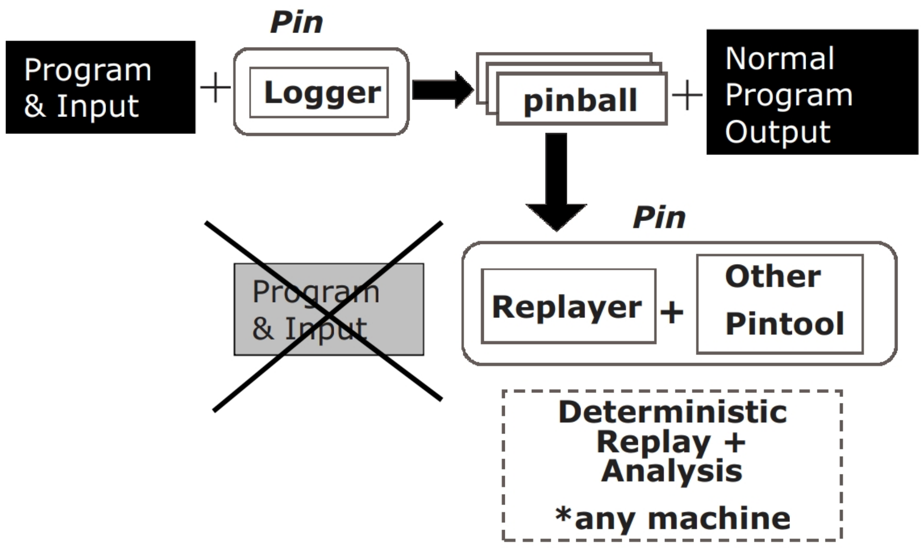


Figure 2.4: PinPlay's architecture. Image taken from [29]

One interesting feature that can be noticed from the figure is that after recording the program execution, PinPlay does not need the program nor the input in subsequently runs, because all the information necessary to replay the execution is stored into the *pinball.*

To be able to remain consistent during subsequent executions of a program, PinPlay needs to log all sorts of details about the execution and later be able to reconstruct and guide the execution to the same path as is was previously recorded. Next, we see a list of aspects which are monitored during the execution:

- **System calls:** During the logging phase, all registers modified by a system call have their value recorded and later, during the replay phase, most system calls are skipped and their behavior is reproduced by modifying the values to match the ones observed during the logging phase.

- **Program binary / shared library:** It might seem obvious that the program itself should not be modified between subsequently replay executions, but what might not be obvious is that all the shared libraries that it depends on should also not change. Freezing all the shared libraries used by the program might not be practical in some environments. Because of this, PinPlay chooses to capture and copy all the code need during the replay (including the shared libraries) and store it into the *pinball*

- **Signals:** Signals are handled similarly as system calls. PinPlay keeps track of in which instruction they occurred; records the state of registers before and after the signal, and during replay it only modifies the values of those registers during that particular instruction to match the ones previously observed.

- **Un-initialized memory variables:** To replicate the same behavior, all variables which are read before being written have their initial value recorded, for later, during the replay initialization have their value restored as they were initially captured
- **Processor specific instructions:** Because PinPlay provides the feature of allowing the logging of a program in one machine and later replay in another, it is also aware of instructions that are CPU specific (such as CPUID). Like signals and system calls, PinPlay records the values inside pertinent registers and restore them during replay.
- **Shared memory access order:** As said earlier, while PinPlay does not guarantee global ordering in the execution of instructions, it needs, at least, to maintain the same access order on shared memory space. To accomplish this, it records the order of accesses to the shared memory during the logging phase and during replay, it watches if the order is respected. If at any point in time it perceives that the order will not be obeyed, it stalls threads or processes until it guarantees the order of access will occur as it was previously recorded.

2.5 Techniques for reducing simulation time

Research in computer architecture often requires a detailed understanding of the behavior of a processor while a program is running.

Many programs have very different behaviors during different parts of their execution that we call phases. At one point they may use memory intensely, at others they may suffer greatly from errors in branch predictions.

To obtain this level of information, researchers often use simulators that model the platform of interest at each cycle executed. Unfortunately, this simulation detail brings with it simulation time penalties, which means that industry benchmarks take months to fully execute.

Further compounding the simulation time situation, it is often necessary to simulate the same benchmark over and over again until researchers find an architecture configuration that has a good balance between consumption, performance, and complexity. For example, often the same program/input pair may be simulated multiple times so that the difference in performance that changing the cache size can bring to a given architecture can be examined.

This problem has not been missed by the academic community, and many researchers have developed techniques that seek to reduce the simulation time [10, 40, 9, 11].

Among the related work, some seek to reduce the simulation time using phase analysis. In our work, we focus on those.

Next, we explain to the reader what *program phase* is, and how it constitutes the basis for the SimPoints methodology.

2.5.1 Program Phases

The way programs behave during execution is often not random. Many studies [37, 17] have shown that programs often engage in repetitive behaviors called phases.

The authors of [37] define phases as the set of intervals (or slices in time) within the execution of a program that has similar behavior, regardless of temporal adjacency.

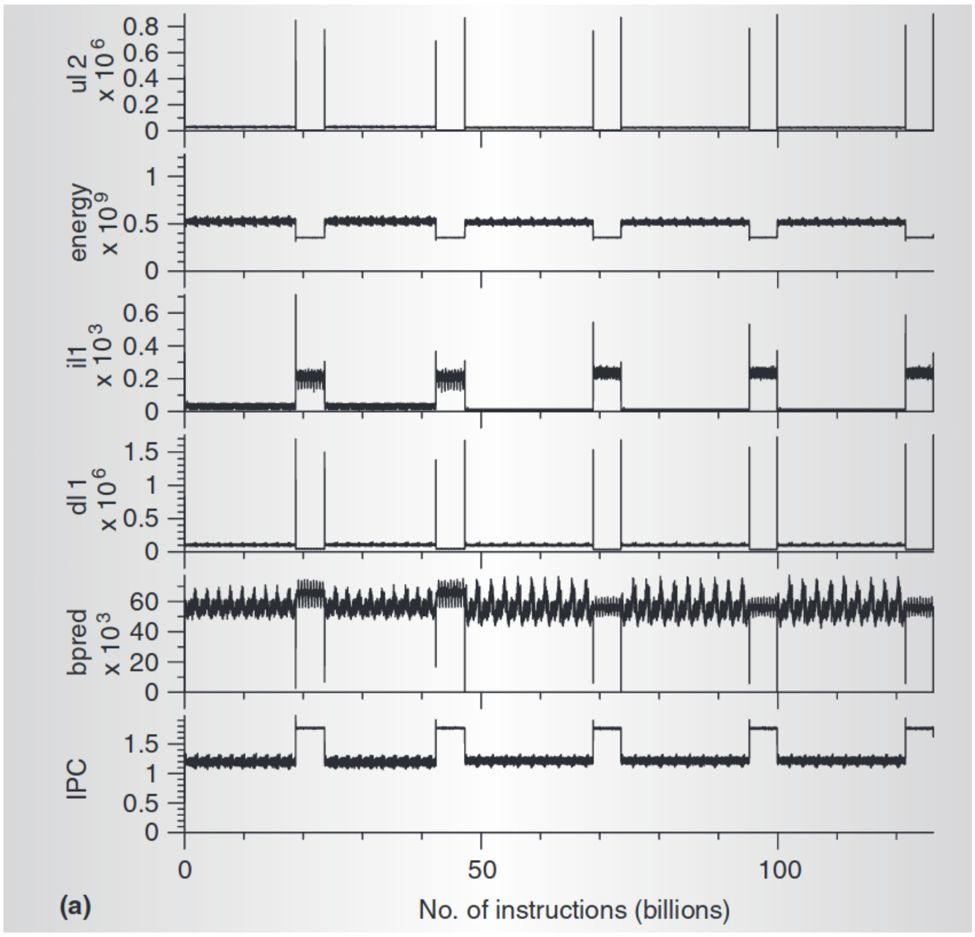


Figure 2.5: Gzip: Plot of some computer architecture metrics over billions of instructions when processing a graphic input. Image taken from [37]

Figure 2.5 shows the variation of some metrics such as CPI, energy consumed and others during gzip program execution. In it, we can clearly see a phase behavior in the program that repeats over time.

A key observation that makes studying program phases important is that any program metric is a direct function of the way a program walks through code while executing [37]. In this way, it is possible to find the phases of a program by examining which regions of code are being executed over time.

2.5.2 SimPoints

One of the techniques proposed to solve the problem of trying to reduce the simulation time required to evaluate platforms is called *SimPoint* [31, 14, 15]. SimPoint intelligently chooses a sample set of the program, which is called *Simulation Points*. These simulation points are chosen in a way that by only simulating them, it is possible to have a good estimate of how the tested platform would perform during the simulation of the whole program.

The *SimPoint* methodology uses grouping algorithms to automatically find repetitive patterns in the execution of a program. By simulating only one representative of each phase of the program, the simulation time can be reduced to minutes rather than weeks implying only a small loss of accuracy.

A key point of the *SimPoint* methodology is that the Simulation Points chosen by the technique are independent of the architecture used for the simulation, thus allowing the same set of Simulation Points to be used for the simulation of many architecture configurations.

In order to make the choice of Simulation Points independent of architecture, the concept of program profiling using a structure called BBV (Basic Block Vectors) has been proposed in [35].

BBV is a vector where each position represents a basic block of the program. Each element of the vector stores the number of times a given basic block has been executed during a predefined interval of instructions and since we are not interested in the absolute value of each position of the vector but in the proportion of the execution of each basic block, normally BBV is normalized by dividing each element by the sum of all elements of the vector. This normalization ensures that the sum of all vector elements is 1, making the comparison of BBV of different interval sizes possible.

Thus, if we slice the program execution into multiple pieces, and store a BBV for each slice of the program, we can compare each slice in regards to where, in the program code, it spends its execution on.

The methodology

SimPoint is a methodology for identifying representative portions of a program. In the methodology, the execution of a program is divided into intervals of an equal number of instructions.

For each interval, a BBV is collected. *SimPoint* then groups the basic block vectors (BBV) using the K-means [16] algorithm. In this grouping, each vector position is plotted as a dimension for the clustering algorithm.

Since we expect the program to have phases (as seen in the previous section), we expect the intervals that correspond to the same phase of the program to be close to each other during the clustering process. Thus, after clustering, if we choose an interval from each cluster, then we have a representative of each phase of the program.

In the SimPoints technique, the interval that is chosen to represent each cluster is the closest one to the centroid of each cluster. These centroid intervals are called Simulation Points (SimPoints) by the technique.

Since the phases of a program have different sizes and since each of these Simulation Points represents a phase of the program, each Simulation Point naturally has different relevance in composing the behavior of the entire program. This way each SimPoint has a weight. This weight is given by the ratio of the number of intervals that the grouping it came from had, and the total number of intervals of the entire program.

Therefore, if you want to predict the CPI value of the entire program by simulating only the Simulation Points, you must make a weighted average of the Simulation Point CPI value considering its weight.

As an example, suppose we have an application that has only 2 basic blocks and when executed with a given input executes 10,000,000 instructions. For illustration purposes, we are going to split it into 10 intervals (1,000,000 instruction intervals each). We chose to illustrate the technique with a hypothetical program that has only two basic blocks so that we can better visualize it.

Initially, the technique will execute the entire program and collect the BBV from each of these intervals. In figure 2.6 each axis represents a basic block and each point marked with a "+" represents one program interval. Note that as we said earlier, the basic block vectors (BBV's) are normalized. Thus the sum of the components of each point on the graph is always 1. For this reason, all points are aligned in this example.

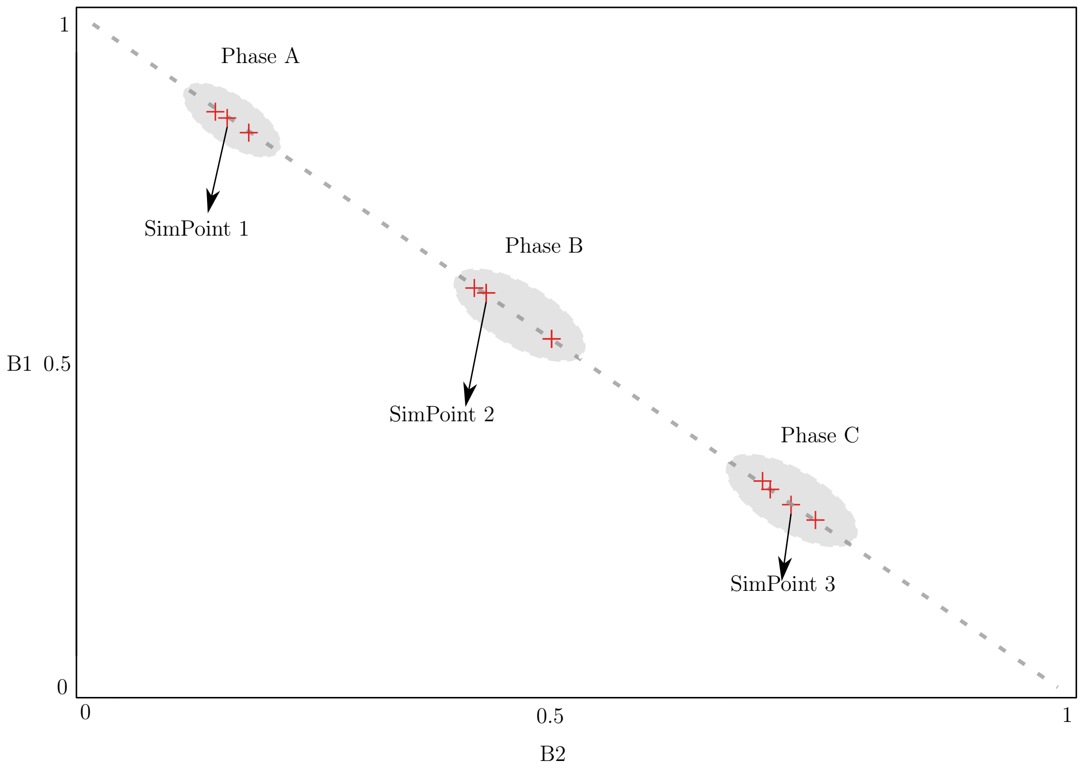


Figure 2.6: Phases A, B, and C generated by the clustering intervals of the same program.

SimPonts technique will use the *k-means* algorithm to automatically find all clusters. The next step is to find, for each group, the interval that is closest to the centroid of the group. This interval will be chosen as the representative of that group and will now be one of the SimPoints.

After knowing the SimPoints of a program, suppose we want to know the expected CPI of a program. The estimated CPI can be calculated with the following formula:

$$CPI_{program} = w_1 * CPI_{SimPoint1} + w_2 * CPI_{SimPoint2} + w_3 * CPI_{SimPoint3}$$

where

$$w_i = \frac{Number\ of\ intervals\ in\ Phase\ i}{Number\ of\ intervals\ in\ the\ whole\ program}$$

For the particular example of figure 2.6, we have:

$$CPI_{program} = 0.3 * CPI_{SimPoint1} + 0.3 * CPI_{SimPoint2} + 0.4 * CPI_{SimPoint3}$$

2.5.3 PinPoints

PinPoints is the result of joining the *SimPoint* technique with the PinPlay framework. In [27] the authors describe how they combined them.

As discussed in 2.5.2 SimPoint methodology proposes a way to estimate the performance of a program. Here the authors present a project capable of automatically generating Simulation Points and also comparing the results of the prediction given by using SimPoints with the whole execution of the program.

To accomplish their goals, the authors take advantage of the PinPlay capabilities of recording and later replaying deterministically the execution while still instrumenting it with the Pin framework.

To compare the results, the authors use the Sniper x86 simulator, which is capable of executing both *pinballs* (recorded executions made by PinPlay) and normal binaries for the x86 platform. It is important to note that Sniper is only capable of running *pinballs* itself because it is built on top of Pin.

Although PinPoints tries to be as automatic as possible, it still needs some parameters which are up to its user to decide. Those are:

- **Interval size:** the number of instructions each interval will have.
- **MaxK used in k-means:** Because SimPoint uses the k-means clustering algorithm, the user needs to provide this parameter. The implication of this parameter is that k-means will never generate more clusters than the value of MaxK, therefore the user needs to at least have an idea of how many phases the program has.
- **Warmup size:** To reduce inaccuracy related to cold cache when simulating a SimPoint, PinPoints offers the user the possibility of running a number of instructions right before starting the SimPoint, just so the cache is not completely cold.

The authors describe PinPoints as being composed of the following steps:

1. PinPoints records the execution of the whole program using PinPlay. The output of this step is a *pinball* of the whole program.
2. The *pinball* generated in the previous step is replayed, and during the replay, the BBV (basic block vectors) for each interval is collected using a pintool.

3. BBVs collected in the last step are used as inputs for the SimPoints technique. The output of this step is a list of intervals chosen as SimPoints.

4. With the information of which intervals are SimPoints, PinPoints replay the whole program pinball again, but this time, during replay, it records the execution of each SimPoint, generating a pinball for each SimPoint. The authors call each of these pinballs a *PinPoint*.

5. They use Sniper to simulate each of the PinPoints as well as the pinball of the whole program. During the Sniper simulation, they collect hardware performance metrics, such as CPI, branch prediction, cache miss and others.

6. In the last step, they use the formula provided by the SimPoint technique and compare the results given by SimPoints with the real values, which are given by the results collected during the Sniper simulation of the whole program pinball.

Chapter 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.

The main goal of this work is to study the challenges of building an easy to use DBI framework, on top of another known open source DBI framework, while discussing the issues concerning the viability of such DBI.

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.

We start this chapter by explaining why we chose DynamoRIO as the foundation for our DBI. After that, we present to the reader a brief overview of how DrPin is structured under the hood. Next, we walk the reader through a small example to illustrate the differences in the ease of use of both Pin's 2 and DynamoRIO's APIs. Following the chapter, we discuss DrPin API compatibility with Pin and walk through some examples demonstrating DrPin executing on both x86-64 and AArch64 architectures. Then, we discuss some performance optimizations that took place during the development of DrPin followed by a performance comparison of DrPin, Pin 2, and DynamoRIO. We finish the chapter by showing how DrPin behaves during the instrumentation of real-world applications.

3.1 DynamoRIO as basis infrastructure

Since Dynamic instrumentation is not easy and a minor bug in its code can result in tremendous instability during execution that are very hard to debug, we wanted to find a battle-proof DBI framework that was actively used, maintained, had a community around it, and was open-source, thus accepting possible bug fixes and improvements as needed.

While studying DBI frameworks it was noticeable that in this arena, besides Pin, two frameworks were dominant: 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. The tool, called memcheck, is so popular that Valgrind authors decided to run it by default if no other tool is specified to the Valgrind engine. This is the reason why many novice C/C++ developers wrongly assume Valgrind is just a memory analysis tool.

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 architect 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 executes.

Therefore, Valgrind offers an API to interact with the application code while it is in this intermediate representation. This makes building tools that work on all Valgrind supported platform easy, but also makes it difficult to match the level of platform details provided by the Pin API, which focuses primarily only on the x86-64 platform.

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 a 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 architect 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.

However, there are drawbacks to our approach: We only have access to DynamoRIO's public interface, meaning we cannot take advantage of any private information that is only available on DynamoRIO's core.

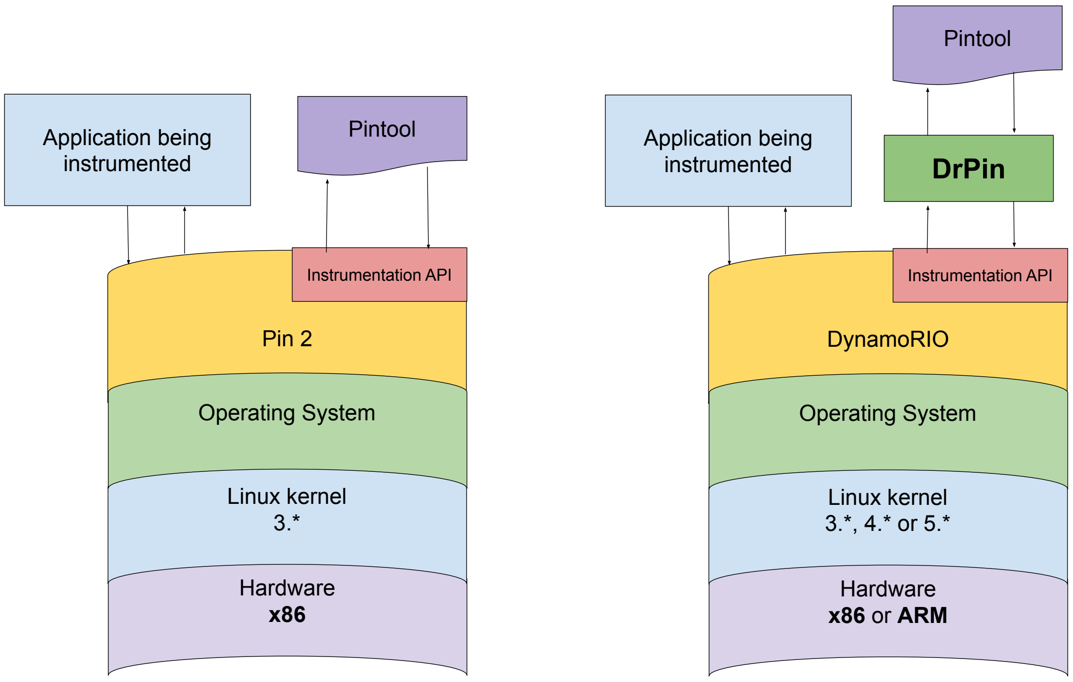


Figure 3.1: DrPin Overview

Figure 3.1 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. 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.

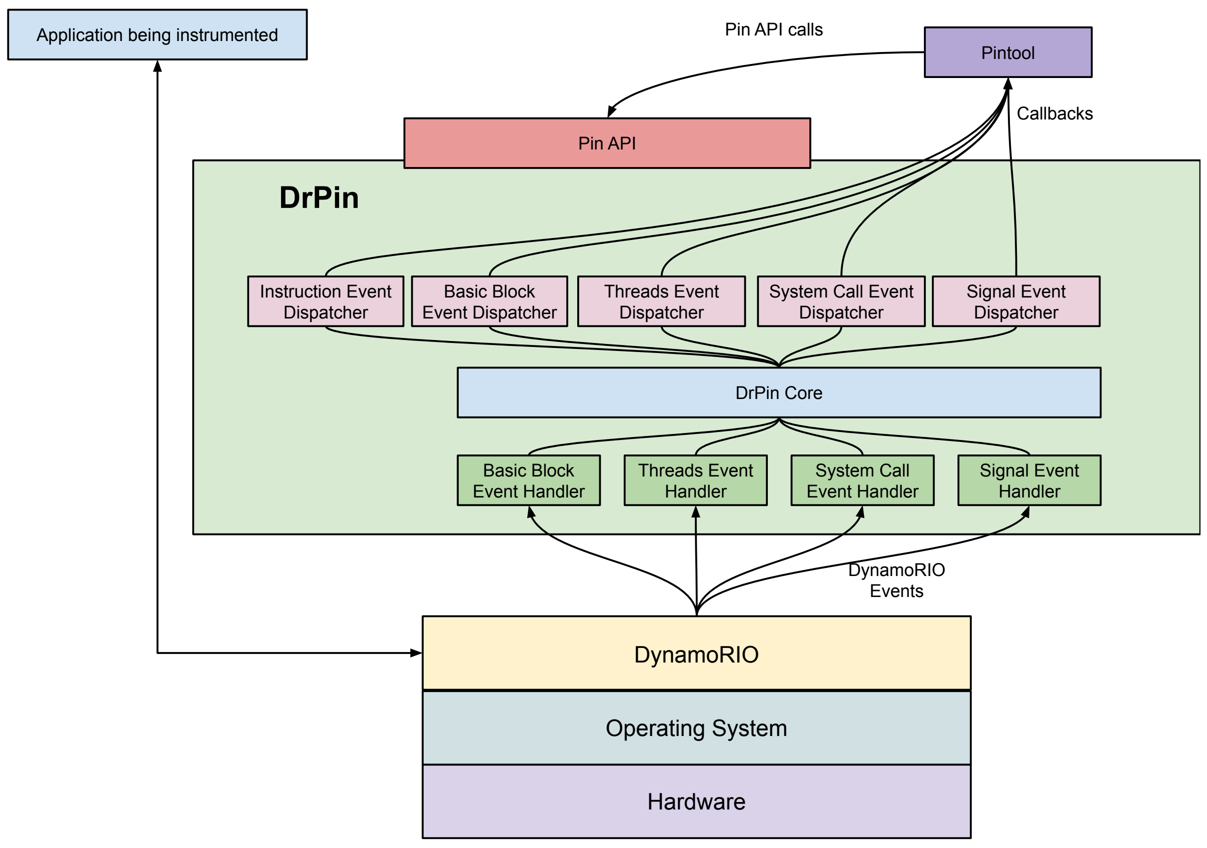


Figure 3.2: DrPin: Arquitecture

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. This 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.

As represented in Figure 3.2, DrPin subscribes to all events DynamoRIO issue 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.

3.3 API friendliness

It is not an accident that Pin 2 became a success right after its launch. It was a DBI framework that, unlike anything before it, had an easy to use API that made instrumenting binary code dynamically accessible to everyone. The key difference in the Pin design that made it so easy to use was the way users were required to interact with it. The interaction was mostly done through callback functions that users registered when they were interested in a particular event about the execution of the application. Parallel to this event-callback mechanism, the core principle in the Pin-way of doing instrumentation is registering an analysis function alongside with the desired runtime information (like a Thread Id, or the value of a particular register, etc.) the users would like to receive as parameters of the analysis function. Putting in another way, users would define analysis functions that expect runtime information as parameters, and Pin would invoke them with the needed information at the proper time during execution.

Because of that, users were able to write their analysis functions in a higher abstraction language (C++) and let pin handle all the hassle related to gathering the information that would be used as input by the analysis function, as well as taking care of not disturbing the execution of the instrumented application.

To better illustrate how much simple it is to instrument and analyze an application using the Pin API, we present listings 3.1 and 3.2. Both listings implement the same analysis: they count the number of instructions executed by a given application.

The first, 3.1, is an excerpt from a pintool. The later, 3.2, is an excerpt from a DynamoRIO client (a client is analogous to a pintool in DynamoRIO's terminology). The full version of both programs can be found and in `https://github.com/luisfernandoantonioli/inscount_pintools`.

```
{...}
static UINT64 icount = 0;

VOID docount() { icount++; }

VOID Instruction(INS ins, VOID *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

VOID Fini(INT32 code, VOID *v)
{
    std::cout << "Count: " << icount << std::endl;
}
{...}

int main(int argc, char * argv[])
{
    if (PIN_Init(argc, argv)) return ErrorInitPin();

    INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(Fini, 0);

    PIN_StartProgram();

    return 0;
}
```

Listing 3.1: Pin: Instruction counter

```
{...}
static uint64 global_count;

static void inscount(uint num_instrs)
{
    global_count += num_instrs;
}

{...}

DR_EXPORT void dr_client_main(client_id_t id, int argc, const char *argv[])
{
    {...}

    if (!droption_parser_t::parse_argv(DROPTION_SCOPE_CLIENT, argc, argv, NULL, NULL))
        DR_ASSERT(false);
    drmgr_init();

    {...}

    dr_register_exit_event(event_exit);
    drmgr_register_bb_instrumentation_event(event_bb_analysis, event_app_instruction,
                                            NULL);

    {...}
}

static void event_exit(void)
{
  /* code to print the number of instructions executed */
  {...}
}

static dr_emit_flags_t event_bb_analysis(void *drcontext, void *tag,
                                         instrlist_t *bb, bool for_trace,
                                         bool translating, void **user_data)
{
    instr_t *instr;
    uint num_instrs;

    {...}

    /* Count group of emulated instructions as only one */
    bool is_emulation = false;
    for (instr = instrlist_first(bb), num_instrs = 0; instr != NULL;
         instr = instr_get_next(instr)) {
        if (drmgr_is_emulation_start(instr)) {
            num_instrs++;
            is_emulation = true;
            continue;
        }
        if (drmgr_is_emulation_end(instr)) {
            is_emulation = false;
            continue;
        }
        if (is_emulation)
            continue;
        if (!instr_is_app(instr))
            continue;
        num_instrs++;
    }
    *user_data = (void *)(ptr_uint_t)num_instrs;

    {...}

    return DR_EMIT_DEFAULT;
}

static dr_emit_flags_t
event_app_instruction(void *drcontext, void *tag, instrlist_t *bb, instr_t *instr,
                      bool for_trace, bool translating, void *user_data)
{
    uint num_instrs;
    drmgr_disable_auto_predication(drcontext, bb);
    if (!drmgr_is_first_instr(drcontext, instr))
        return DR_EMIT_DEFAULT;
    if (user_data == NULL)
        return DR_EMIT_DEFAULT;
    num_instrs = (uint)(ptr_uint_t)user_data;
    dr_insert_clean_call(drcontext, bb, instrlist_first_app(bb), (void *)inscount,
                         false /* save fpstate */, 1, OPND_CREATE_INT32(num_instrs));
    return DR_EMIT_DEFAULT;
}
```

Listing 3.2: DynamoRIO: Instruction counter

We chose the instruction counter pintool to compare both APIs because it is usually a canonical example in DBI frameworks documentation, and therefore, are easily found inside the official documentation.

Not only the full version of 3.1, the pintool, has fewer lines of code, it is also simpler and abstracts away much of the hassle involved in the process of instrumenting and analyzing a particular program.

There are several reasons for why the DynamoRIO client has more code. First, DynamoRIO has support for running multiple clients at once (what they call "chaining clients") as well as emulating instructions (adding a series of instructions to the binary that should be treated as a single instruction. This feature makes possible to simulate a hypothetical instruction). Because of that, it is common for DynamoRIO clients to be aware of the possibility of running in the application code after another client already passed through the application code adding emulation instructions.

Second, Pin's API is more granular, providing specific functions for all sorts of events related to the lifecycle of the application, while DynamoRIO provides fewer events and expects its clients to iterate through provided data structures.

As a result, simple pintools that are only interested in a specific aspect of the execution of an application can be written with very few lines of code using the Pin API.

Although the difference in verboseness of both API's is already clear in listings 3.1 and 3.2, the difference will rapidly grow as the complexity of the pintool grows as well.

The main reason is the difference between `INS_InsertCall` (from the Pin API) and `dr_insert_clean_call` (from the DynamoRIO). While both provide means for the pintool writer to insert calls anywhere in the instrumented code to an analysis function, the Pin variant packs a lot more functionality. `INS_InsertCall` provides many conveniences, like invoking the analysis function passing runtime information as argument, while the latter can only provide fixed arguments defined at instrumentation time. Therefore, the user of `dr_insert_clean_call` have to add a lot of code into its analysis function to get the information needed by his analysis.

Another source of complexity in DynamoRIO API is the fact that it does not try to hide the complexity of the engine from its user. This becomes evident in listing 3.2 where the tool writer is required to return a constant (in this case `DR_EMIT_DEFAULT`) in many places to indicate to DynamoRIO how it would like to emit code into the code cache. Knowing how the code cache works is important for DynamoRIO users that care about performance, but might be another entry lever barrier for beginners with little understanding of dynamic instrumentation. Even though Pin also have API functions that let its users to interact with the code cache, these interactions are usually reserved only to advanced API functions.

3.4 Pin API Compatibility

One of the goals of DrPin is to progressively support the most used Pin 2.0 API functions, aiming to be a drop-in replacement for Pin in most cases, thus allowing most of the pintools made for the original Pin to work with DrPin without any modifications.

As briefly mentioned in section 2.3.3, Pin instruments the application by using a just-in-time compiler, but in contrast to other popular just-in-time compilers that use bytecode, Pin uses the native binary as input for the JIT compiler.

Pin intercepts the application at the first instruction and using the pintool provided by the user, it decides what instructions should be added to the native code. One important aspect of the process is that Pin does not instrument the whole binary at once.

Instead, it instruments a trace at a time. Pin calls "a trace" a sequence of instructions that may start anywhere and finishes at an unconditional branch or by other heuristic, such as when the trace already has a certain number of conditional branches.

During the instrumentation of a trace Pin also changes the addresses of control flow instructions to ensure that it does not lose control over the application when the instrumented code is being executed.

After this piece of code (trace) is instrumented, it then moves the newly compiled code into what Pin calls the "code cache". Pin never executes any application code that has not been moved into the code cache.

Lastly, Pin transfers the control to the code that resides in the code cache, allowing the application to run (now instrumented) and waits until the execution of the program returns to it, starting again its cycle of instrumenting, sending to the code-cache, and resuming the execution.

Every time Pin regains control of the process execution, it interacts with the pintool by possibly calling many callback functions that were registered by the pintool, thus giving the pintool the opportunity to instrument the next piece of code.

Therefore we can logically split the time during the execution of a program under Pin into two major parts: When Pin is reading the original binary and generating the new binary code, and when the newly generated code is actually running. This logical separation is important to understand because all callbacks registered by the pintool will only be called during this phase.

A common pattern for pintools is to use the instrumentation phase to choose places to insert calls to analysis functions (using API functions like `INS_InsertCall` and `BBL_InsertCall`). The analysis functions are also defined inside the pintool, but will be executed during the second phase, when the actual instrumented application is running.

A key difference between Pin and many other DBI's, is that not only it allows the insertion of calls to these analysis functions (as DynamoRIO also does with its `dr_insert_clean_call` API function), but also it supports sending runtime information to these analysis functions as well, making them much more powerful.

To illustrate such functionality, imagine that one can write an analysis function that will be called before every conditional branch is executed, and as a parameter, this analysis function will receive if that particular branch instruction (that will be executed next) will be taken or not. Not many DBI frameworks offers this functionality out of the box.

This is one of the many runtime informations that an analysis function can receive and Pin would do what is necessary to provide the information and not disrupt the application execution.

The Pin API is composed of a total of 549 functions that can be divided into eight main categories:

- Instruction
- Basic Block
- Trace
- Section
- Routine
- Image
- Register
- Other Utilities

The instruction API partition is composed of functions that either register callback functions for the events that involve instructions only (such as when Pin encounters an instruction during the compilation phase) or functions that gives information about a particular instruction (such as the address of the instruction, or how many operands does the instruction have). All these API functions use the `"INS_"` prefix.

Very similar to the functions related to instructions, basic block API functions operate on a basic block granularity and are also composed of events related to basic blocks, as well as information about them. These API functions use the `"BBL_"` prefix.

Trace API functions operate very similarly to the basic block functions, but instead of basic blocks, they operate on traces.

Similarly, section and image API functions also have their own prefixes (`"SEC_"`, and `"IMG_"` respectively) and offer functionalities and events related to their granularity.

Lastly, what we categorized as other utilities are functions for several purposes that usually have the `"PIN_"` prefix. They are used for all sorts of tasks, from initializing Pin data structures, to getting a system call argument during a particular place of the execution.

Because the Pin API is very extensive, DrPin strategy is to progressively support it, expanding the support one function at the time, increasing the number of supported pintools. Naturally, some functions of the API are more popular than others, and implementing those would allow us to support a bigger set of existing pintools.

The order in which API functions were implemented was guided by the tradeoff between popularity and engineering effort of each API function.

To guide us in the process of defining the popularity of each API function, we used a mix of pintools. The mix is composed of the pintools that comes in a folder bundled with the original Pin distribution called *SimpleExamples* as well as two other popular pintools: Allcache and ZSim.

The *SimpleExamples* directory that comes with Pin that is meant to be a diverse set of pintools to illustrate to pin users the vast Pin API. It is composed of 22 pintools that uses a total of 117 different API functions. In order to visualize better the most popular API functions, we built figure 3.3. The figure is a histogram, that shows for each API function, how many pintools inside the *SimpleExamples* folder uses it.

As expected, both `PIN_StartProgram` and `PIN_Init` are present on all pintools, since they are mandatory.

DrPin implemented all API functions from this list, as all of them are widely used by pintool writers.

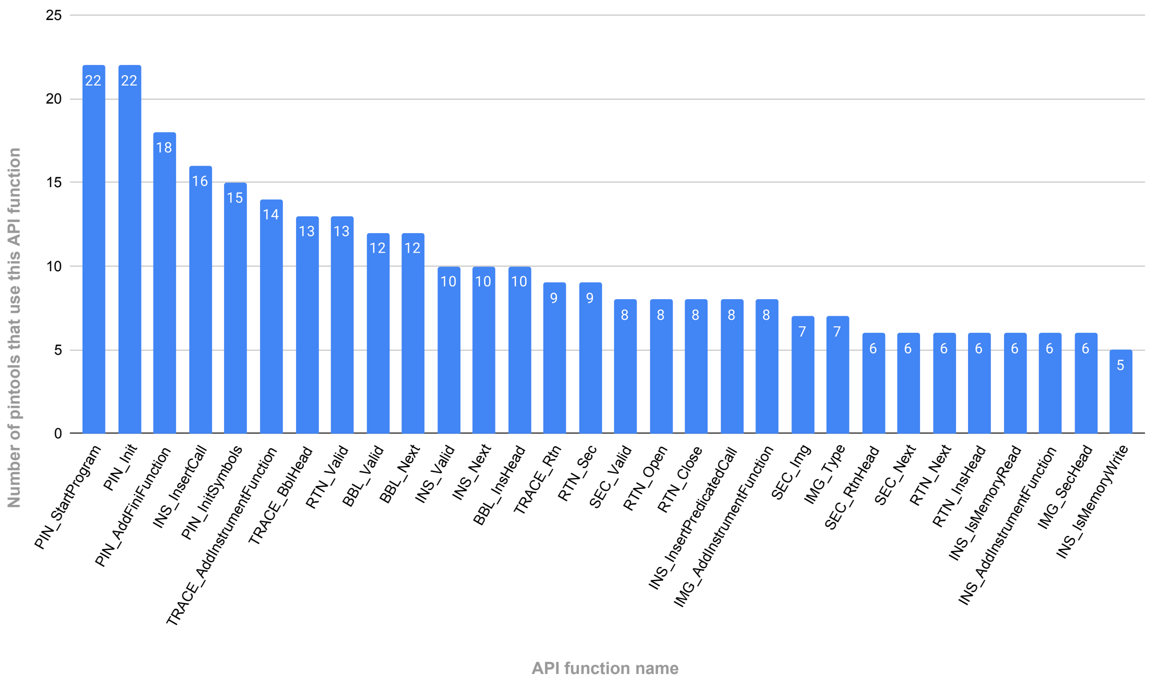


Figure 3.3: Top 30 most executed API functions by the SimpleExample pintools

Table 3.1 shows all pintools present in the folder, describing their purpose and whether or not DrPin currently support them.

Table 3.1: SimpleExamples pintools

Pintool	Description	Supported
Calltrace	Traces call instructions	yes
Catmix	Category mix profiler	yes
Coco	Code coverage analyzer	yes
Dcache	Cache simulator	yes
Edgcnt	Edge Profile collector	yes
Emuload	Load emulator	yes
Extmix	Static and dynamic instruction mix profiler	yes
Fence	Runtime program text modification guard	no
Icount	Dynamic instructions counter	yes
Ilenmix	Static and dynamic instruction lenght mix profiler	yes
Inscount2_mt	Multi-thread dynamic instruction counter	yes
Jumpmix	Jump instructions profiler	yes
Ldstmix	Dynamic register/memory mix profiler	yes
Malloctrace	Malloc calls tracer	no
Opcodemix	Static and Dynamic opcode mix profiler	yes
Oper-imm	Prints information on immediate operands	yes
Pinatrace	Memory access tracer	yes
Regmix	Static and Dynamic register mix profiler	yes
Regval	Demonstrates the use of PIN_GetContextRegval API	yes
Topopcode	Realtime opcode mix profiler	yes
Toprtn	Realtime top routines	yes
Trace	Instructions tracer	yes

The *SimpleExample* pintools guided us in the beginning of DrPin development, but because its primary purpose is to familiarize pintool writers with the vast API, besides the first 30 most popular functions, the lists starts to become too sparse to have a real connection with the actual popularity of the other API functions.

Because of that, we looked for other popular pintools, and guided DrPin API development around them, targeting their support.

Allcache is an interesting pintool written by Artur Klauser that does functional simulation of TLB and cache hierarchies. It is distributed alongs with Pin. The pintool uses 11 API functions and currently runs with no problems on DrPin. Because none of the API used are specific to the x86 architecture, it can also run on the other architectures supported by DrPin.

Another famous pintool is Zsim [32]. Zsim is a fast and accurate multicore x86-64 simulator that focuses on simulating memory hierarchies and large heterogeneous systems. In essence, it is a large pintool that uses Pin to construct and update internal models about the simulated system.

Zsim uses a total of 80 different Pin API functions, which is more than most pintools do. Because it uses several libraries, it does not currently work on Pin 3. For instance, some of Zsim dependencies like `libelf`, `libhdf5`, and `libpthread` are incompatible with PinCRT.

DrPin support for Zsim is not fully stable, but DrPin can already successfully run the examples shown in its *readme*.

Zsim case is not alone. Many pintool's writers were not capable of porting their tools to use PinCRT even when willing to re-write their tools. One example is the authors of the Hybrid Verifier Cross-Platform Verification Framework for Instruction Set Simulators

[13]. Their pintool uses network sockets to send and receive data, but when they tried to use the PinCRT sockets API they were surprised to find out that even though PinCRT has a `sendTo` function in its API, it does not have implemented a function to receive data through a socket yet.

At the time of this writing, DrPin already supports 83 API functions. The complete list of all supported API functions can be found in appendix A.

3.5 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 3.3. The pintool collects the mnemonic of every instruction executed and at the end of the execution, displays a histogram of the 30 most executed mnemonics.

```
#include "pin.H"
// ... other includes

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);
  // Inserts mnemonic into histogram if it does not exists yet
  if (mnemonicIndex == -1) {
    // Pushing new mnemonic to the end of the mnemonic histogram
    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_30_mnemonics();
}

int main(int argc, char *argv[]) {
  if (PIN_Init(argc, argv)) {
```

```
36     return 1;
37   }
38 
39   INS_AddInstrumentFunction(instrumentInstruction, 0);
40 
41   PIN_AddFiniFunction(fini, 0);
42 
43   PIN_StartProgram();
44 
45   return 0;
46 }
```

Listing 3.3: Pintool: Top 30 mnemonics

Although the pintool presented in listing 3.3 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.

The pintool 3.3 is very concise. It maintains an array of counters (one per mnemonic) and updates it before every execution of an instruction.

The pintool starts by calling the `PIN_Init` function. This function is present in the beginning of every pintool, and is used by DrPin to initialize all its internal data structures. Next, in the line 39, we register a callback function (`instrumentInstruction` into DrPin using `Ins_AddInstrumentFunction` API function. After this registration, whenever a new instruction is found (during the compilation phase of DrPin, as described in 3.4), the callback function will be called, giving us the opportunity to instrument the instruction.

Our callback function, defined in line 15 is very short as well. We use the `INS_Disassemble` API function to disassemble the current instruction, and use a simple `string` operation to extract the mnemonic of the instruction. After that, we check if that particular mnemonic is already present in our array of mnemonic counters, and if not, we initialized its position with zero.

Then, we ask DrPin to insert a call before that particular instruction to our analysis function called `"incrementMnemonicCount"`. Because our analysis function is used by every instruction, it needs to know what mnemonic will be executed next, so it can update the corresponding counter. We could pass a string with the mnemonic name to the analysis function, but an index of the corresponding counter suffice.

At the end of the main function, we also register a callback function called “`fini`” to be called whenever the application being instrumented finishes its execution. We use the “`fini`” function to sort and print the results accumulated in the mnemonic counters.

To exemplify the functionality of such pintool, we are going to instrument 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.

```
mov        : 150408
cmp        : 47419
add        : 44746
jz         : 41268
test       : 35868
jnz        : 28868
push       : 23115
lea        : 22110
pop        : 20314
movzx      : 20208
jmp        : 17214
and        : 15889
sub        : 11479
xor        : 10581
data16     : 7991
jnbe       : 7420
call       : 7353
ret        : 7349
shr        : 6243
jbe        : 3593
shl        : 3565
jb         : 3435
or         : 3413
setnz      : 3331
pcmpeqb    : 3083
jnb        : 3018
setz       : 2232
movdqa     : 2163
movsxd     : 1740
vpcmpeqb   : 1732
```

Listing 3.4: Top 30 mnemonics: x86-64 output

```
add        : 162417
ldr        : 152696
subs       : 134900
orr        : 117405
ldrb       : 67643
str        : 64406
stp        : 49024
cbz        : 48960
ldp        : 44350
cbnz       : 38602
movz       : 36697
sub        : 33877
b.eq       : 33858
b.ls       : 29333
csinc      : 29227
adrp       : 25163
b.hi       : 24800
and        : 24365
b          : 23781
eor        : 16346
b.ne       : 15984
ubfm       : 15539
ands       : 14064
ret        : 12958
bl         : 11768
ccmp       : 11642
b.cc       : 10767
strb       : 10586
br         : 10365
udiv       : 10218
```

Listing 3.5: Top 30 mnemonics: AArch64 output

DrPin executed 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.

Listings 3.4 and 3.5 shows the output of the pintool on the x86 system and on the AArch64 one respectively, running GNU `ls`.

Another class of pintools that are reasonably architecture-independent are the ones that perform analysis on semantics that are closer to source code constructs. To exemplify this class of pintools, we created the pintool present in listing 3.8 (the full pintool source code can be found in `https://github.com/luisfernandoantonioli/callgraph_pintool`).

```
#include "pin.H"
{...}

FILE * fp;

class Graph {
  public:
    void AddEdge(std::string& source, std::string& target){
        {...}
    }
    void GenerateCallgraph(FILE* f){
        {...}
    }
};

Graph graph;

VOID createEdge(ADDRINT callerAddr, ADDRINT calleeAddr ){
  std::string callerName = RTN_FindNameByAddress(callerAddr);
  std::string calleeName = RTN_FindNameByAddress(calleeAddr);
  graph.AddEdge(callerName, calleeName);
}

VOID instrumentInstruction(INS ins, VOID *v) {
  if(INS_IsDirectBranchOrCall(ins)){
    ADDRINT ins_addr = INS_Address(ins);
    ADDRINT target_addr = INS_DirectBranchOrCallTargetAddress(ins);

    IMG ins_img = IMG_FindByAddress(ins_addr);
    IMG target_img = IMG_FindByAddress(target_addr);
    if(IMG_Valid(ins_img) && IMG_Valid(target_img) && IMG_IsMainExecutable(ins_img) &&
    IMG_IsMainExecutable(target_img)){
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)createEdge, IARG_ADDRINT, ins_addr,
    IARG_ADDRINT, target_addr, IARG_END);
    }
  }
}

VOID fini(INT32 code, VOID *v) {
  fp = fopen ("./callgraph.dot","w");
  graph.GenerateCallgraph(fp);
  fclose (fp);
}

int main(int argc, char *argv[]) {
  if (PIN_Init(argc, argv)) {
    return 1;
  }

  PIN_InitSymbols();

  INS_AddInstrumentFunction(instrumentInstruction, 0);

  PIN_AddFiniFunction(fini, 0);

  PIN_StartProgram();

  return 0;
}
```

Listing 3.6: Callgraph pintool

The pintool 3.6 creates a graph where the functions defined in the source code of the application are *nodes*, and the calling relation between them are *graph edges*. The name of the pintool is `callgraph` because it generates a call graph of the instrumented application. To operate correctly, the pintool needs to know the names of the functions defined in the source code, therefore it expects the application to have debug information embedded into the binary.

On Linux, the DWARF [8] format is the most popular debug format, and can be easily embedded into the binary application. Currently, DrPin only supports the Dwarf debugging format.

The pintool operation is quite simple. It inspects every instruction with the help of the `instrumentInstruction` function. When it finds an instruction that is a `call` instruction or a direct `branch`, it instruments the binary to insert a call to the `createEdge` function. `instrumentInstruction` also does not instrument instructions that are in other shared libraries.

It is important to notice that createEdge will only be executed for that particular instruction if at runtime, the execution of the program passed through that part of the binary code. Another aspect is that `createEdge` receives two addresses as input, the caller address and callee address, so before adding the edge into the graph, it first needs to ask DrPin for the name of the function that contains that address.

To illustrate the execution of the Callgraph pintool, we will use the C program defined in listing 3.7. Although the program is not particularly useful, it contains a lot of functions calls and fits well the purpose of illustrating the pintool

```c
#include<stdio.h>

int fib0(){
    return 0;
}
int fib1(){
    return 1;
}
int fib2(){
    return fib1() + fib0();
}
int fib3(){
    return fib2() + fib1();
}
int fib4(){
    return fib3() + fib2();
}
int fib5(){
    return fib4() + fib3();
}
int fib6(){
    return fib5() + fib4();
}

void print(int value){
    printf("The 6th fibonacci number: %d\n", value);
}

int main(int argc, char const *argv[])
{
    int value = fib6();
    print(value);
    return 0;
}
```

Listing 3.7: Callgraph demo application

The output of the pintool is presented in listing 3.7. Conveniently, the pintool outputs its result in the graphviz dot file format [12]. Using graphviz Linux program, we can transform the text output into figure 3.4. The number near the edges of the graph represents how many times a particular edge of the graph was activated during the execution of the 3.7 program.

```


digraph callgraph {
 "main" -> "fib6" [label=1]
 "fib6" -> "fib4" [label=1]
 "fib4" -> "fib2" [label=2]
 "fib2" -> "fib0" [label=5]
 "fib2" -> "fib1" [label=5]
 "fib4" -> "fib3" [label=2]
 "fib3" -> "fib1" [label=3]
 "fib3" -> "fib2" [label=3]
 "fib6" -> "fib5" [label=1]
 "fib5" -> "fib3" [label=1]
 "fib5" -> "fib4" [label=1]
 "main" -> "print" [label=1]
}
```

Listing 3.8: Callgraph dot file output

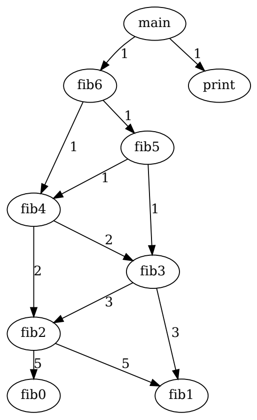


Figure 3.4: Callgraph visual output

Although DrPin makes easy for its users to write architecture-independent pintools, 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 ilustrate 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 archtectures is still a valuable asset for the pintool writer.

3.6 Peformance Optimizations

Although DynamoRIO and Pin have a lot of similarities under-the-hood, on the API level, Pin values simplicity and ease-of-use. 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 parameters it receives:

- **INS ins**: an object that represents an instruction of the application.
- **IPOINT action**: an enum to control where in the neiborhood of the instruction, the call to the analysis function will be placed.
- **AFUNPTR funptr**: a pointer to the analisys function.

- `...`: zero or more parameters 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 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 analisys function. To work around this, DynamoRIO clients requests the needed runtime information inside the analisys 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 (instruction addresses, function names and etc).

To mitigate this problem, we optimized DrPin to inspect the requested function arguments and when it is detected 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, but also preserved Pin most valuable functionalities.

3.7 Performance Comparison

DBI frameworks have always paid additional attention to their performance. Performance, in their context, is not a luxury, but a necessity. DBI tools usually interacts with every single instruction executed by an application, and a minor overhead in its operation can add up, resulting in an increase in the application execution time of many orders of magnitude.

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. Therefore, there is no single dimension or metric that can overall describe the performance of a DBI framework. DBI frameworks not only vary greatly their performance when running different applications, but more importantly, have completely different performances depending on the type of analysis tool that is built with them.

To better illustrate the performance characteristics of DrPin, we focused our attention on two distinct scenarios commonly found in the DBI world.

In the first scenario, we want to quantify the intrinsic overhead created by the DBI framework engines. As the reader might be wondering, all the infrastructure built inside DBI engines to provide the ability to instrument a binary program (as the JIT compiler and the code-cache) does not come for free, even if no instrumentation is inserted into the application.

To test the intrinsic overhead created by DrPin, and compare it with other DBI frameworks (DynamoRIO and Pin), we decided to measure the slowdown of running the SPECint 2006 benchmark on these frameworks with an empty pintool. To reduce the variance caused by different operating systems and libraries, we also executed all frameworks in the same machine with the turbo boost disabled. The machine had the following specifications:

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

Unfortunately, because Pin 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/DynamoRIO because GCC 4.8.5 did not fully support all the C++11 features used in DynamoRIO and DrPin codebase.

Figure 3.5 shows the slowdown of each SPECint 2006 program for each DBI framework. From figure 3.5 we observe that DrPin did not significantly increase the slowdown when compared with DynamoRIO. This is really good, because although DrPin subscribes to every application lifecycle event available in the DynamoRIO framework, this did not translated into a big overhead during runtime.

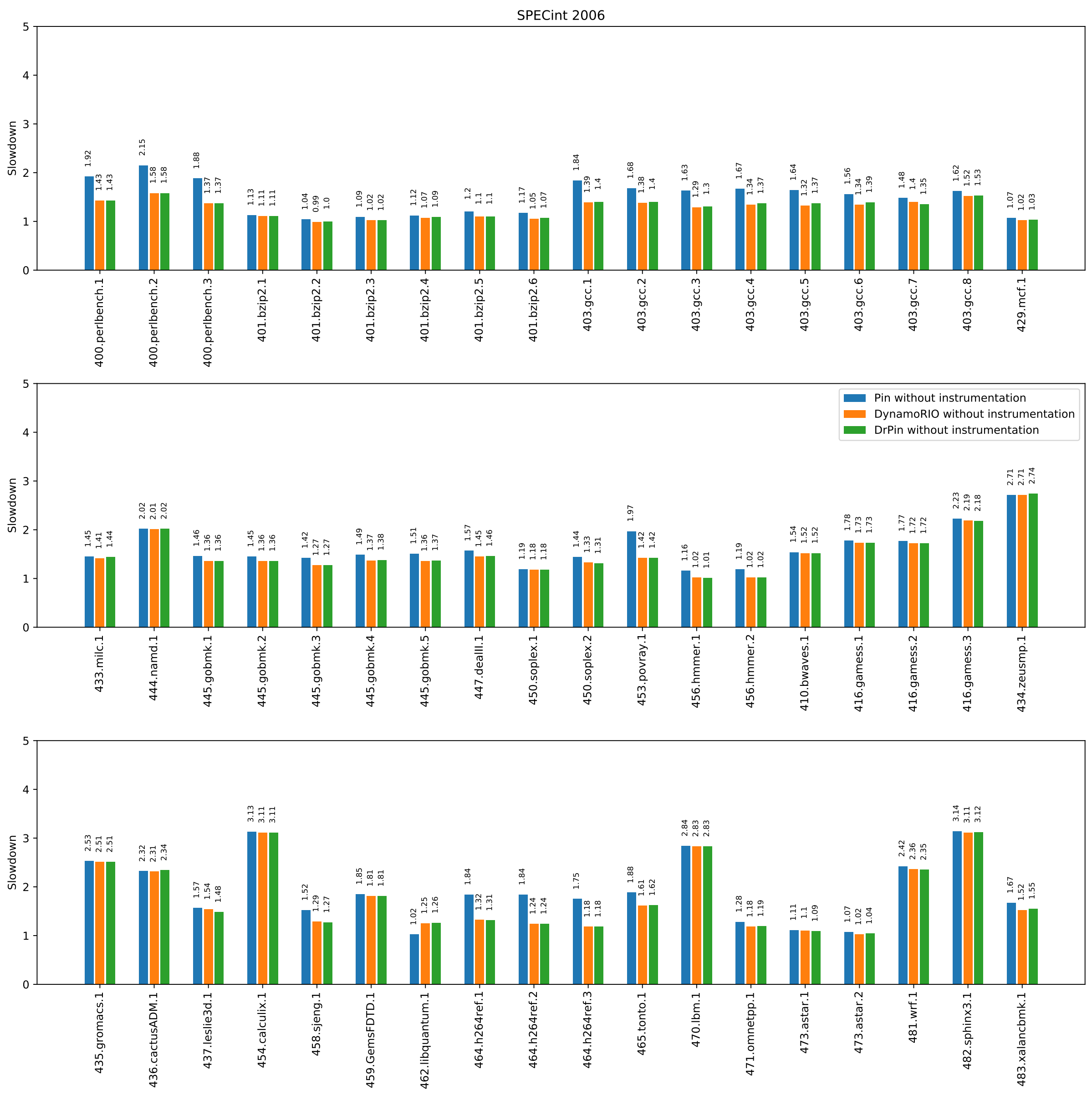


Figure 3.5: SPECint 2006: Intrinsic execution slowdown

Another characteristic shown in figure 3.5 is that both DynamoRIO, Pin, and DrPin have very similar intrinsic slowdowns. We believe this comes from the fact that both DynamoRIO and Pin have very similar architectures, being built on top of a JIT compiler, and also use a code cache to remove unnecessary re-instrumentations.

As said at the beginning of this section, there is no upper limit to the slowdown that a dynamic instrumentation tool can cause, as the analysis functions can be of any complexity. Therefore to compare the slowdown of the DBI frameworks, we chose the instruction counter pintool. This pintool 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.

The pintool used by both Pin and DrPin is the same presented in listing 3.1, and the client used by DynamoRIO is the same that was presented in listing 3.2. We also used the SPECint 2006 benchmark for this test, as well as the same machine described in the previous slowdown test.

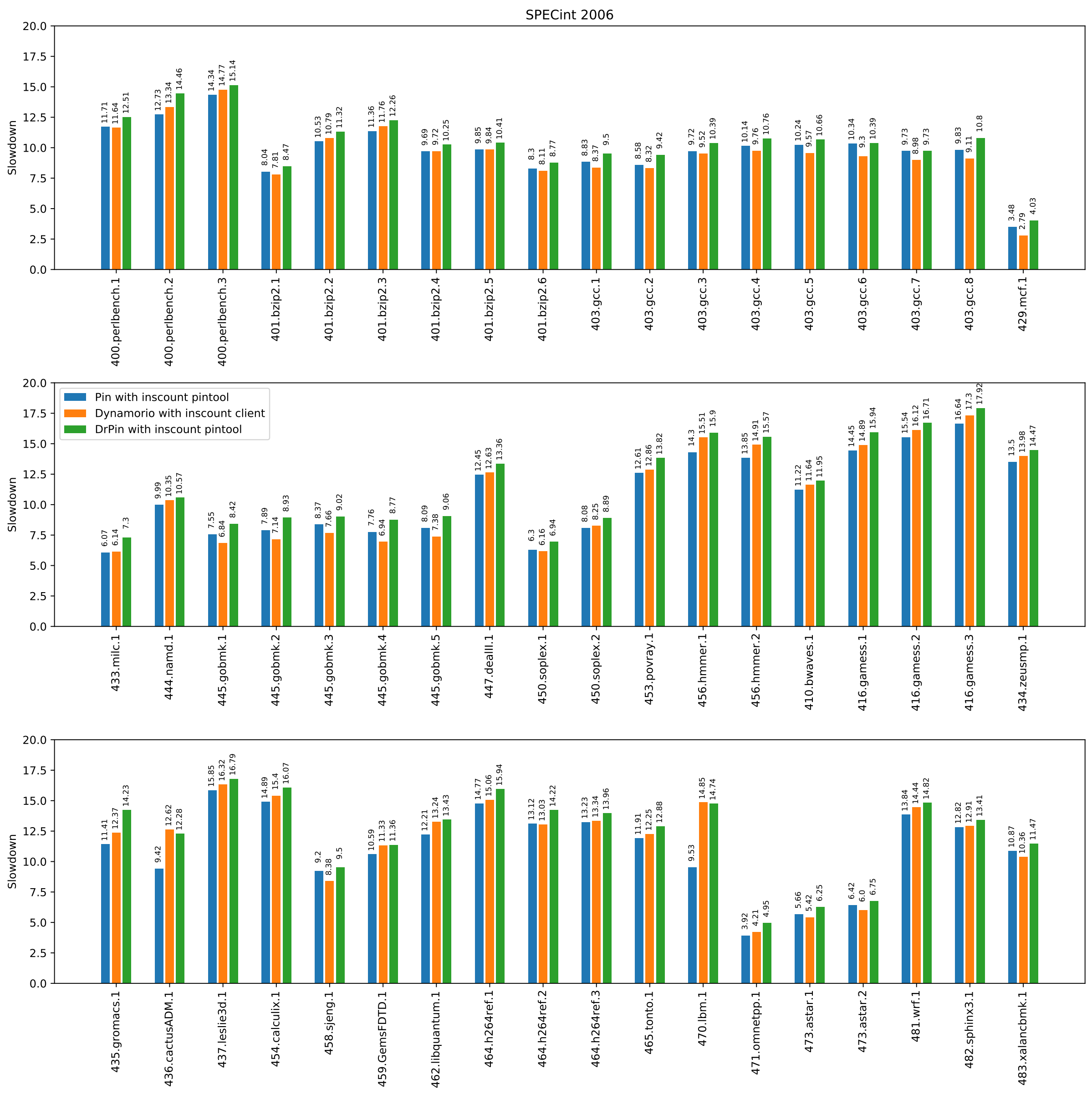


Figure 3.6: SPECint 2006: Instruction count execution slowdown

In figure 3.6 we see that in this scenario the performance of all three DBI frameworks is close, with Pin being a little faster than DynamoRIO, and DynamoRIO faster then DrPin. On average, DrPin was 10% slower than Pin and 11.6 times slower than the native execution.

We already expected DrPin to be a little slower than DynamoRIO, as it is natural since DrPin uses DynamoRIO under the hood. What was not so expected was DynamoRIO being a little slower than Pin.

DynamoRIO inscount pintool can be made faster at the cost of losing compatibility with other architectures, by using x86 specific API functions to increment the instruction counter inline, removing the need to call a function to increment the global counter. We did not go through this path because our interest was not to make the fastest possible inscount pintool, but rather, to understand the slowdown generated by a typical dynamic instrumentation tool built by these frameworks.

In section 3.6 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 doesn't 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.

To complete our performance evaluation, we wanted to show the implications this aspect in the slowdown of DrPin execution. To abuse this weakness of DrPin, 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. The pintool is described in listing 3.9.

```
#include "pin.H"
#include <iostream>
using namespace std;

UINT64 cond_br_counter;
UINT64 cond_br_taken_counter;

VOID instr_branch( BOOL branch_taken ) {
  cond_br_counter ++;
  if (branch_taken){
    cond_br_taken_counter ++;
  }
}

VOID Instruction(INS ins, VOID *v)
{
  if ( INS_Category(ins) == XED_CATEGORY_COND_BR ){
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)instr_branch, IARG_BRANCH_TAKEN, IARG_END
    );
  }
}

VOID PrintStatistics(INT32 exit_code, VOID *v){
  cout << "=====================================================" << endl;
  cout << "Number of executed conditional branches: " << cond_br_counter << endl;
  cout << "Number of taken conditional branches: " << cond_br_taken_counter << endl;
  cout << "=====================================================" << endl;
}

int main(int argc, char *argv[])
{
    if( PIN_Init(argc,argv) )
    {
        cout << "initialization invalid" << endl;
    }

    cond_br_counter = 0;

    INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(PrintStatistics, 0);

    PIN_StartProgram();

    return 0;
}
```

Listing 3.9: Conditional Branch Taken pintool

We believe that this pintool is near the worst-case scenario of DrPin compared to Pin, because of all API functions that were implemented in DrPin that 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, because we think it is enough to show the slowdown.

Table 3.2 shows DrPin being two orders of magnitude slower than Pin for the giving pintool.

Table 3.2: SPECint 2006: GCC Slowdown

	Slowdown	
Programs	**Pin**	**DrPin**
403.gcc.1	7.79	455.86
403.gcc.2	7.25	426.45
403.gcc.3	8.19	526.58
403.gcc.4	8.51	493.9
403.gcc.5	8.67	542.02
403.gcc.6	8.39	529.09
403.gcc.7	8.48	540.04
403.gcc.8	7.85	484.08

Even though DrPin provides Pin's 2 API to its users, one is not only limited to use the Pin API. DrPin users can mix both Pin and DynamoRIO's API. Therefore, in the case presented in table 3.2, the pintool writer could take advantage of DynamoRIO's API and thus reduce the slowdown by directly using specific DynamoRIO's API functions to only retrieve the runtime information it needs for its analysis

3.8 Instrumentation of real-world applications

The advantages of using dynamic instrumentation of binaries increases as the size and complexity of application increases as well. Modern large applications are built from multiple libraries and developers most of the time do not have all the source code of all the pieces that forms the application. Analyses only with the source code might not be enough to understand the application behavior.

But even dynamic instrumentation tools face difficulties dealing with massive modern applications such as web browsers.

As of May of 2020, nether Pin, Valgrind, nor DynamoRIO can run the Google Chrome Web Browser consistently without crashing. Such applications, as Google Chrome, are a result of millions of lines of code that exposes DBI frameworks to edge cases that have not yet been properly handled.

During the development of DrPin, a major concern was to try to maintain the same level of reliability provided by DynamoRIO.

To verify how DrPin behaves compared to Pin instrumenting a large application, we created a simple pintool, described in listing 3.10, which reports the number of concurrent threads being executed in the application. There is nothing especial about this specific pintool, except that we are going to use it to instrument a large application.

As said earlier, browsers are a good example of modern large application which is everywhere. Unfortunately, we already know that both Pin and DynamoRIO currently have trouble running Google Chrome. As a result, we chose Mozilla Firefox for our tests.

We used Firefox version 66.0.3 in our tests because it is present in Ubuntu 14.04 package repository. The machine used in our tests has the following specification:

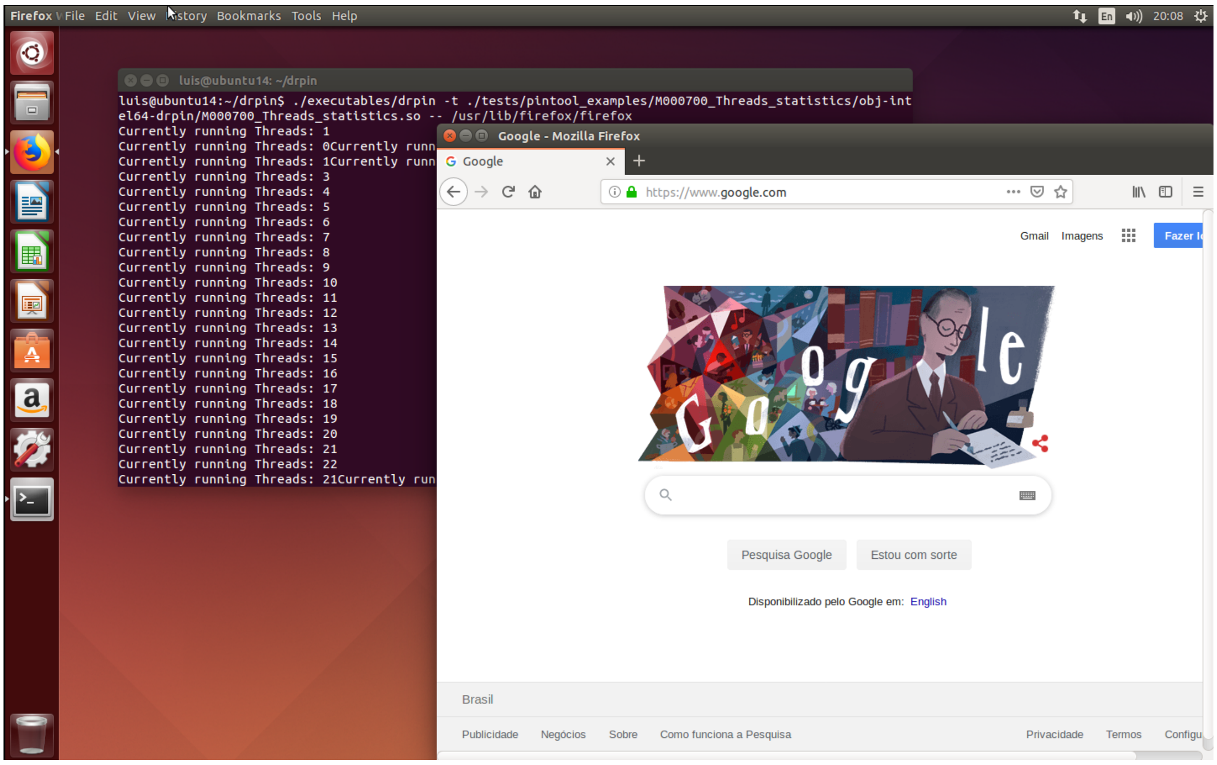
- **OS**: Ubuntu 14.04.6 LTS
- **Kernel**: 3.19.0-25-generic
- **Compiler**: GCC 7.4.0 for DrPin and GCC 4.8.5 for Pin
- **Processor**: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz

This specification was mostly chosen as a result of the Pin limitations.

```
#include "pin.H"
#include <iostream>

using namespace std;

UINT64 cur_num_of_threads;

VOID ThreadStartHandler (THREADID threadIndex, CONTEXT *ctxt, INT32 flags, VOID *v){
  cur_num_of_threads ++;
  cout << "Currently running Threads: " << cur_num_of_threads << endl;
}

VOID ThreadStopHandler (THREADID threadIndex, const CONTEXT *ctxt, INT32 code, VOID *v){
  cur_num_of_threads --;
  cout << "Currently running Threads: " << cur_num_of_threads;
}

int main(int argc, char *argv[]) {
    if( PIN_Init(argc,argv) ) {
        cout << "initialization invalid" << endl;
    }

    cur_num_of_threads = 0;

    // Register the ThreadStartHandler function to be called
    // when a thread starts in application code
    PIN_AddThreadStartFunction(ThreadStartHandler, 0);

    // Register the ThreadStopHandler function to be called
    // when a thread exits in application code
    PIN_AddThreadFiniFunction(ThreadStopHandler, 0);

    PIN_StartProgram();

    return 0;
}
```

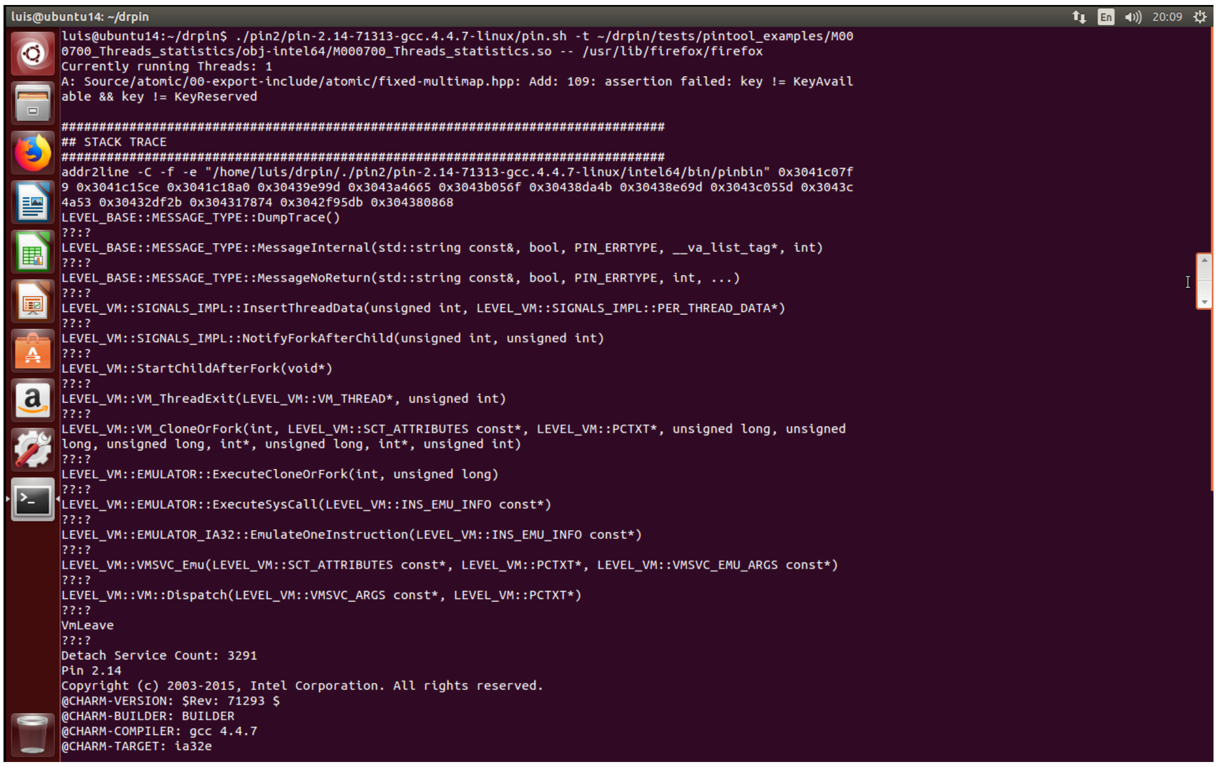
Listing 3.10: Threads statistics pintool

The pintool described in listing 3.10 maintains a global counter that is incremented whenever a new thread starts in the application and is decremented when a thread is stopped. Figure 3.7 shows a screenshot of DrPin instrumenting Firefox with the pintool 3.10, and figure 3.8 shows Pin trying to run the same pintool to instrument Firefox.



```
luis@ubuntu14:~/drpin$ ./executables/drpin -t ./tests/pintool_examples/M000700_Threads_statistics/obj-int
el64-drpin/M000700_Threads_statistics.so -- /usr/lib/firefox/firefox
Currently running Threads: 1
Currently running Threads: 0Currently runn
Currently running Threads: 1Currently runn
Currently running Threads: 3
Currently running Threads: 4
Currently running Threads: 5
Currently running Threads: 6
Currently running Threads: 7
Currently running Threads: 8
Currently running Threads: 9
Currently running Threads: 10
Currently running Threads: 11
Currently running Threads: 12
Currently running Threads: 13
Currently running Threads: 14
Currently running Threads: 15
Currently running Threads: 16
Currently running Threads: 17
Currently running Threads: 18
Currently running Threads: 19
Currently running Threads: 20
Currently running Threads: 21
Currently running Threads: 22
Currently running Threads: 21Currently run
```

Figure 3.7: DrPin running firefox



```
luis@ubuntu14:~/drpin$ ./pin2/pin-2.14-71313-gcc.4.4.7-linux/pin.sh -t ~/drpin/tests/pintool_examples/M00
0700_Threads_statistics/obj-intel64/M000700_Threads_statistics.so -- /usr/lib/firefox/firefox
Currently running Threads: 1
A: Source/atomic/00-export-include/atomic/fixed-multimap.hpp: Add: 109: assertion failed: key != KeyAvail
able && key != KeyReserved

################################################################################
## STACK TRACE
################################################################################
addr2line -C -f -e "/home/luis/drpin/./pin2/pin-2.14-71313-gcc.4.4.7-linux/intel64/bin/pinbin" 0x3041c07f
9 0x3041c15ce 0x3041c18a0 0x30439e99d 0x3043a4665 0x3043b056f 0x30438da4b 0x30438e69d 0x3043c055d 0x3043c
4a53 0x30432df2b 0x304317874 0x3042f95db 0x304380868
LEVEL_BASE::MESSAGE_TYPE::DumpTrace()
??:?
LEVEL_BASE::MESSAGE_TYPE::MessageInternal(std::string const&, bool, PIN_ERRTYPE, __va_list_tag*, int)
??:?
LEVEL_BASE::MESSAGE_TYPE::MessageNoReturn(std::string const&, bool, PIN_ERRTYPE, int, ...)
??:?
LEVEL_VM::SIGNALS_IMPL::InsertThreadData(unsigned int, LEVEL_VM::SIGNALS_IMPL::PER_THREAD_DATA*)
??:?
LEVEL_VM::SIGNALS_IMPL::NotifyForkAfterChild(unsigned int, unsigned int)
??:?
LEVEL_VM::StartChildAfterFork(void*)
??:?
LEVEL_VM::VM_ThreadExit(LEVEL_VM::VM_THREAD*, unsigned int)
??:?
LEVEL_VM::VM_CloneOrFork(int, LEVEL_VM::SCT_ATTRIBUTES const*, LEVEL_VM::PCTXT*, unsigned long, unsigned
long, unsigned long, int*, unsigned long, int*, unsigned int)
??:?
LEVEL_VM::EMULATOR::ExecuteCloneOrFork(int, unsigned long)
??:?
LEVEL_VM::EMULATOR::ExecuteSysCall(LEVEL_VM::INS_EMU_INFO const*)
??:?
LEVEL_VM::EMULATOR_IA32::EmulateOneInstruction(LEVEL_VM::INS_EMU_INFO const*)
??:?
LEVEL_VM::VMSVC_Emu(LEVEL_VM::SCT_ATTRIBUTES const*, LEVEL_VM::PCTXT*, LEVEL_VM::VMSVC_EMU_ARGS const*)
??:?
LEVEL_VM::VM::Dispatch(LEVEL_VM::VMSVC_ARGS const*, LEVEL_VM::PCTXT*)
??:?
VmLeave
??:?
Detach Service Count: 3291
Pin 2.14
Copyright (c) 2003-2015, Intel Corporation. All rights reserved.
@CHARM-VERSION: $Rev: 71293 $
@CHARM-BUILDER: BUILDER
@CHARM-COMPILER: gcc 4.4.7
@CHARM-TARGET: ia32e
```

Figure 3.8: Pin trying to run firefox

DrPin had no problems and correctly instrumented Firefox. Pin, however, could not. Although pin outputs a limited stack trace showing what went wrong, since Pin source code is closed, the community cannot investigate further the causes, and possibly contribute with a fix.

DrPin's goal was to bring together the easy-of-use of Pin 2, without the restrictions of Pin 3, while still being able to execute in modern operating systems. DynamoRIO turned out to be a solid base for DrPin, providing a stable experience across all supported architectures. An interesting feature that came from using DynamoRIO under the hood was the possibility for the pintool writers to also use DynamoRIO API functions if they wish to do so, further expanding the spectrum of possibilities for a given pintool. DrPin source code can be found in `https://github.com/luisfernandoantonioli/drpin` and we encourage contributions and enhancements by the community.

Chapter 4

SimPoints Among Multiple Program Inputs

Besides the study of how to make dynamic binary instrumentation, which resulted in the construction of DrPin, also explored a little the application of dynamic binary instrumentation in other fields of computer science.

In particular, we studied and expanded the simpoints technique that uses information extracted through dynamic binary instrumentation to make predictions about the performance of a given hardware.

As discussed in section 2.5.2, the SimPoints is an effective methodology for automatically searching and finding representative pieces of the execution of a program, which in turn is very useful when trying to research, prototype and test new ideas in the computer architecture field. The technique allows researchers to have an estimate of how the newly designed platform would perform under real-world benchmarks without having to wait long simulation times, which is normally the case when dealing with real-world benchmarks like `SPECint 2006`.

One of the contributions of our work was to propose and validate an extension to the SimPoints technique.

The *SimPoint* methodology automatically finds a small set of *Simulation Points* that represent the complete execution of a program for a given input. Thus the analysis performed by the methodology is made for each program/input pair separately.

In this work we seek to extend the methodology to perform analysis of multiple inputs of the same program at the same time, thus seeking to find *Simulation Points* that are representative for more than one input.

If the same Simulation Point is used to characterize more than one input, we decrease the total number of Simulation Points needed to simulate a set of program inputs without losing the ability to simulate each input individually. This way we get the hardware metrics like CPI, cache miss, and branch misprediction for each input individually. By reducing the number of Simulation Points required to simulate a set of inputs, we also reduce the simulation time required to simulate all inputs.

Because the PinPoints project (discussed during section 2.5.3) is commonly used to validate the SimPoints technique [24, 27], we also chose to use it to validate our extension of the technique. Unfortunately, we cannot use DrPin to replace Pin inside the PinPoints project. The reason is that PinPoints relies entirely in the PinPlay framework, and as discussed in section 2.4.1, the PinPlay framework has its source code closed, therefore to use DrPin as the basis for the PinPoint project it would be required to re-implement PinPlay from scratch.

We start the chapter by presenting the reader with a formal description of the problem that SimPoints try to solve and also the description of the problem that our extension of the technique tries to solve.

Next we explain how we can leverage the existing SimPoints technique with a few modifications to expand its effectiveness in reducing the required simulation time spent executing a given program with a set of inputs.

To end the chapter we present to the reader a set of experimental evaluations of our technique, comparing it with the original technique, regarding many aspects such as accuracy and number of required representative regions.

4.1 Problem Statement

Let P be any program and $P_D = (i_1, i_2, ..., i_n)$ the instructions that were executed by P for a given fixed input D, or simply its *instruction trace*. Also, consider the collection of all the static basic blocks of P, let $(BB_1, BB_2, ..., BB_r)$ be that collection. Given m, the SimPoint methodology divides P_D in disjoint subsets of size m. Let $I = (I_1, I_2, .., I_l)$ where $l = \lceil \frac{n}{m} \rceil$, the sets resulting from this division. After the nomenclature established by Sherwood *et al.* , we call these sets *instruction intervals*. For each interval $I_k \in I$, a basic block vector BBV_k is generated, such that each position $BBV_k(j)$ represents the number of times the basic block BB_j was executed in the interval k Let $intervals(P, e, m)$ be a function that takes a program P, an input e, and an integer m and returns the set I we described above.

The main objective of the SimPoint technique is to find a partition of I such that each element of this partition contains intervals with similar BBVs and, as consequence, present a similar architectural behavior. Let $(A_1, A_2, ..., A_k)$ be a partition of I. Each set A_i contains intervals of a program phase. The first step consists in selecting a representative interval (SimPoint) from each phase for simulation. A partition of size k implies k SimPoints. Let $(sp_1, sp_2, .., sp_k)$ be the representatives of each one of the phases. Each result of the execution metric of a SimPoint of a phase A_i is weighted by w_i according to the phase coverage size, *i.e.*:

$$w_i = \frac{|A_i|}{|I|} \tag{4.1}$$

If we take, for instance, CPI as a metric to be evaluated using the SimPoint methodology, the CPI of the complete execution can be estimated by the following equation:

$$CPI(P, e) \approx w_1 \times CPI(sp_1) + w_2 \times CPI(sp_2) + ... + w_k \times CPI(sp_k) \tag{4.2}$$

Now, consider the following extension of the SimPoint technique: Let E_P be the set of input of program P, *i.e.*, $E_P = (e_1, e_2, ..., e_n)$ and let I' be the set resulting of the union of all the input intervals in E_P, *i.e.*, $I' = \bigcup_{e \in E_P} \mathit{intervals}(P, e, m)$. This work finds a partition of the set I', clustering all the intervals from all inputs. The objective is to find all distinct phases of P considering the inputs of E_p. We are looking for all equivalent phases of execution common to every input.

Additionally, this work also finds a weight matrix W in which the lines represent the inputs of E_P. and the columns represent the chosen SimPoints. Thus, $W(i)(j)$ represent the weight of the SimPoint j for the input i.

$$W = \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \dots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array}$$

Once again, if we take CPI as the metric, the complete execution of P for the inputs in E_P can be estimated using k SimPoints as:

$$\begin{pmatrix} CPI(e_1) \\ CPI(e_2) \\ \vdots \\ CPI(e_n) \end{pmatrix} \approx \begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \\ e_n \end{array} \begin{array}{c} \begin{array}{cccc} sp_1 & sp_2 & \cdots & sp_k \end{array} \\ \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,k} \\ w_{2,1} & w_{2,2} & \dots & w_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \dots & w_{n,k} \end{pmatrix} \end{array} \times \begin{pmatrix} CPI(sp_1) \\ CPI(sp_2) \\ \vdots \\ CPI(sp_k) \end{pmatrix} \tag{4.3}$$

A trivial partition of I' can be obtained using the SimPoint methodology individually for each input. In this case, values of the line $W(e)$, $e \in E_P$ are zero for the SimPoints of the inputs in $E_P \setminus e$. To illustrate this case, take for instance a program P with an input set $E_P = \{e_i, e_j, e_k\}$. Considering that $SP(e_i) = (sp_A, sp_B, sp_C)$, $SP(e_j) = (sp_D, sp_E)$ and $SP(e_k) = (sp_F, sp_G)$, the matrix W would be:

$$W = \begin{array}{c} \\ e_i \\ e_j \\ e_k \end{array} \begin{array}{c} \begin{array}{ccccccc} sp_A & sp_B & sp_C & sp_D & sp_E & sp_F & sp_G \end{array} \\ \begin{pmatrix} 21\% & 43\% & 36\% & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 38\% & 62\% & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 77\% & 23\% \end{pmatrix} \end{array}$$

The trivial matrix produces the same results of the technique presented by [36]. However, this work clusters all intervals of every input, enabling us to find different phases for different inputs. In other words, a subset of a partition of I' may contain distinct input intervals. The objective is to find the phases presented by an input set and choose a representative interval for each one of these phases. This means that a SimPoint can be used to estimate the execution of two or more inputs.

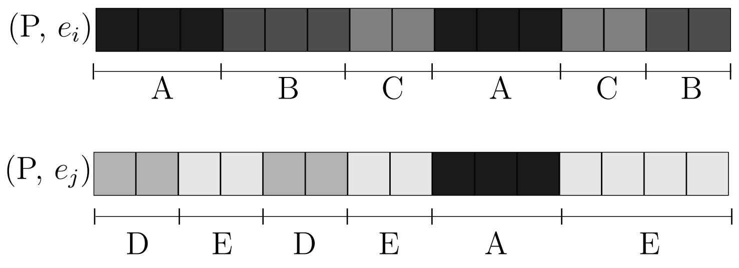


Figure 4.1: Phases of program P with inputs e_i e ej

Figure 4.1 illustrates our approach. Vertical lines show the division in intervals. The input i presents 15 intervals such that: 6 intervals were clustered in phase A, 5 intervals in phase B and 4 intervals phase C. The input j also resulted in 15 intervals clustered in phases D, E, and A. Therefore, 3 input intervals of the input j present similar behavior of the intervals of phase A of the input i, which results in the creation of a single phase A. For this example, the matrix W would have the form:

$$W = \begin{array}{c} \\ e_i \\ e_j \end{array} \begin{array}{c} \begin{array}{ccccc} sp_A & sp_B & sp_C & sp_D & sp_E \end{array} \\ \begin{pmatrix} 40\% & 33\% & 27\% & 0 & 0 \\ 21\% & 0 & 0 & 26\% & \%53 \end{pmatrix} \end{array}$$

For this particular example, the result of a SimPoint execution metric of phase A has distinct values for inputs i and j.

4.2 Multiple-Input Phase Classification

Our approach to find every distinct phase from a program P for an input set E_P employs the same comparison metric and clustering methodology described by [36]. For each interval $I_k \in I'$ a basic block vector BBV_k is generated in a way such that each position $BBV_k(j)$ represents the number of times the basic block BB_j was executed in the interval k. Then, the `k-means` clustering algorithm divides the intervals into phases based on the Euclidian distance among BBVs. Later, a single interval, the one closest to the cluster center (centroid), is chosen to represent each phase. Finally, a detailed simulation is executed using each SimPoint and properly weighted.

In the original methodology, the interval cluster size determines the weight of each phase. On the other hand, in our version the weight of each SimPoint for a given input e is defined by the number of input intervals of e contained by cluster c. Figure 4.2 shows a clustering example for three inputs which resulted in phases A, B, and C along with the chosen SimPoints (centroids) for each phase. In this example the weight of *SimPoint A* for *Input 1* is proportional to the number of intervals of *Input 1* present in cluster A, the weight of *SimPoint A* for *Input 2* is proportional to the number of intervals of *Input 2*, and so on.

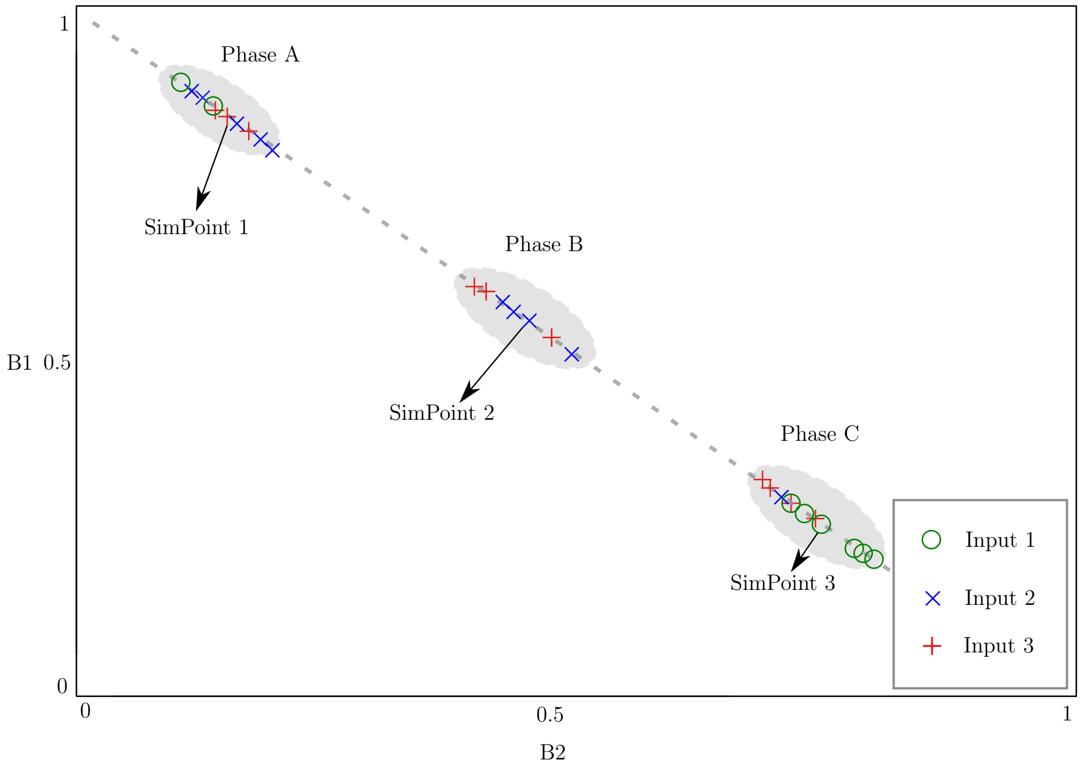


Figure 4.2: Phases A, B, and C generated by the clustering of distinct input intervals of a same program.

Let $(A'_1, A'_2, ..., A'_k)$ be a partition of the set I'. Each A'_i represents a distinct phase presented by the program by the execution of one or more inputs of E_P. Formally, $w(i)(j)$ which represents the weight of the SimPoint j for the input i can be defined as:

$$w(i, j) = \frac{|intervals(P, i, m) \cap A_j|}{|intervals(P, i, m)|} \tag{4.4}$$

Because the SimPoint methodology requires information about the basic blocks executed in each program, and many workloads require a complex execution environment that cannot easily be reproduced on any simulator, PinPoint [28] was introduced as a tool that addresses such problems. Its main purpose is to provide an implementation of the SimPoint methodology on top of the Pin and PinPlay.

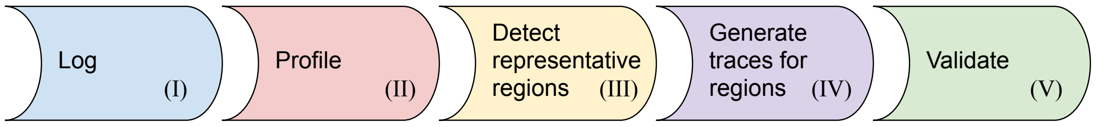

Log (I)
Profile (II)
Detect representative regions (III)
Generate traces for regions (IV)
Validate (V)

Figure 4.3: PinPoint major execution steps

As illustrated in figure 4.3, the PinPoint execution is composed of five major steps. PinPoint automatically (i) logs a program execution, (ii) profiles the application, (iii) detects representative regions, (iv) generates traces for the regions and (v) validates the results using hardware performance counters. In addition, because these five steps are well defined, they were implemented as five distinct programs inside PinPoint framework.

For the purposes of this study, the PinPoint framework has been modified to find SimPoints among multiple program inputs. Steps (i), (ii) and (iv) were used without modification. First, steps (i) and (ii) are run for all inputs of a program. Step (ii) outputs a `.bb` file such that each line represents an execution interval and stores the number of times each basic block was executed during that interval. We then merge all the `.bb` file generated for the multiple inputs of a program. We also keep a record of the number of intervals of each input. We then apply (iii) on the merged `.bb` file.

Step (iii) creates the `.simpoints` and `.weights` files. Each simulation point in the `.simpoints` file contains the number of intervals from the first interval to reach the start of the simulation point. Based on this information and the number of intervals of each input, we are able to infer the input each interval belongs to. The `.weights` are the percentage of intervals of execution being represented by each simulation point. Since the SimPoint methodology was run on the merged BBVs those weights are useless. PinPoint also provides the information about the phase each interval has been assigned to. Based on this information we weight each SimPoint for every input according to Equation 4.4.

Next, we apply (iv) on every input to the generated SimPoints. Finally, in (v) we evaluate the accuracy of the SimPoints by running the whole program and the SimPoint regions using Sniper (a x86 simulator to which PinPoint was integrated) [5].

4.3 Experimental Evaluation

We analysed our proposal using SPECint 2006 reference programs that have multiple inputs: `perlbench`, `bzip2`, `gcc`, `gobmk`, `hmmer`, `h264ref`, and `astar`. We used the PinPoint framework to collect all the basic block vector profiles and generate the representative regions. We evaluate our methodology using the micro-architectural simulator Sniper. After identifying program phases, we used Sniper to compute CPI for SimPoint regions and the whole program. The baseline micro-architectural model used was `nehalem-lite` which is included along with the simulator software.

We compared our technique to the original SimPoint technique, *i.e.*, we applied it to each input separately. When running PinPoint, the user has to specify three parameters: an interval size, a warm-up size and an upper limit on the number of cluster $maxK$, which is essentially the maximum number of distinct phases to be selected by SimPoint. Thus, if a program has N inputs, it will produce a maximum of $N \times maxK$ SimPoints.

To fairly compare the two SimPoint methods (single input and multiple inputs) we used the same SimPoint configurations of interval and warmup for both techniques. We set the interval size to 35 million instructions and warm up to 1 million instructions. We limited SimPoint's maximum number of clusters to 30 for the single input methodology. As outlined in Section 4.2, we found a cluster of the intervals from multiple inputs, based on the merged BBVs from all inputs, using the same clustering algorithm from the original methodology. Therefore, we also have to pick a `maxK` for this clustering. In our technique, for a program with N inputs, we tested two values for $maxK$: $30 \times N$ and $20 \times N$.

The reasoning behind this choice comes from the fact that $maxK = 30$ was chosen in the original technique, Therefore choosing $maxK = 30$ x N enables that if no input shares phases with one another, we still have the same expected level of accuracy as the original technique. Because we expect at least some phases to be shared between distinct inputs, we also tested for $maxK = 20$ x N, to see how the technique would behave with a little more pressure to form fewer clusters.

4.3.1 Experimental Results

To evaluate our approach, we consider three main aspects: (i) the total number of SimPoints, (ii) the difference in precision between our approach and the original technique, and (iii) phase equivalence between the multiple inputs. The first aspect is important because it directly relates to the simulation time, while the second determines the feasibility of our approach. The third aspect, on the other hand, gives insights into the behavior of the programs when we vary the inputs and how those changes influence the generation of SimPoints.

4.3.2 Comparison of the Number of Simulation Points

The number of simulation points is a useful information to estimate the amount of simulation time required by both techniques. SimPoint's clustering algorithm generally picks fewer simulation points than $maxK$ (upper limit) because it usually finds a good phase characterization with fewer clusters. Figure 4.4 shows the number of simulation points generated for each of program in SPECint 2006 with more than one input.

Figure 4.4 shows that our technique is able to find a phase characterization with fewer clusters than SimPoint's original methodology. On average, our technique, compared to the original technique, used 36% fewer SimPoints for $maxK = 30$ and 47% fewer SimPoints for $maxK = 20$ (32% fewer on average). This indicates that it can find a good phase characterization clustering of the intervals from multiple inputs of a program with fewer clusters. As a result, this suggests that inputs do share phases, otherwise the number of simulation points would be close to the sum of all SimPoints applied separately.

4.3.3 Comparison of Errors in CPI

Figure 4.5 shows the CPI error obtained by comparing a complete run to the results obtained from the SimPoint for single input and SimPoint for multiple inputs. The average error for a single input (original approach) is 5.30%; The average error in the analysis using multiple program inputs (our approach) is is 5.36% and 5.74% for $maxK = 30$ and $maxK = 20$ respectively. This implies that both techniques have similar errors. Therefore, this analysis suggests that we can reduce the number of simulation points used by a set of inputs of a program and get similar accuracy.

Even though the total number of SimPoints used in all inputs was lower, the number of SimPoints taken into consideration to estimate the results of each input was higher. This difference in the number of SimPoints does not change the total simulation time and offsets a potential increase in the error rate that could be caused by the use of more general

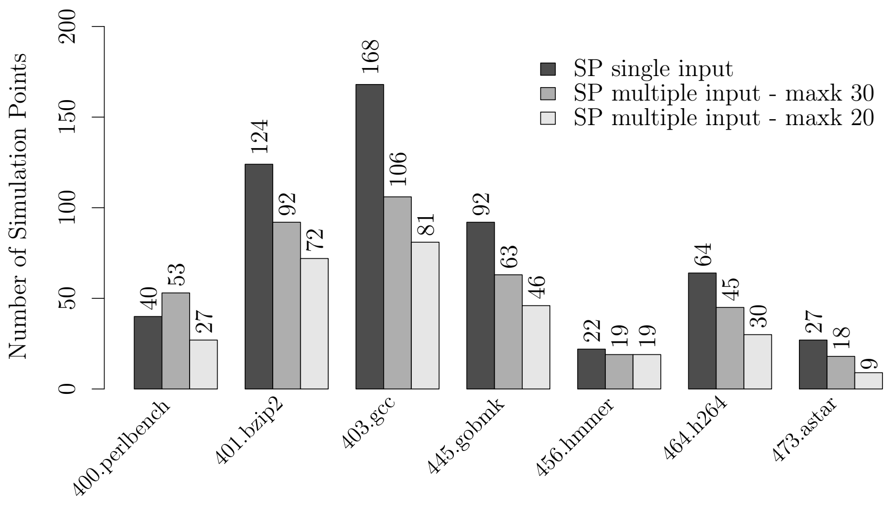


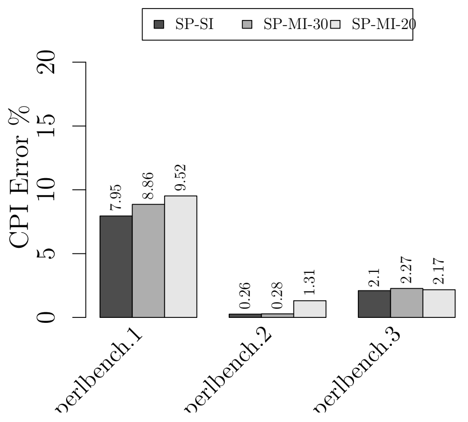
Figure 4.4: Number of Simulation Points

SimPoints instead of more specific ones. Our experimental results show that the average CPI estimate error, when compared to the original approach, is 0.5% for $maxK = 20$ and negligible (0.06%) for $maxK = 30$.

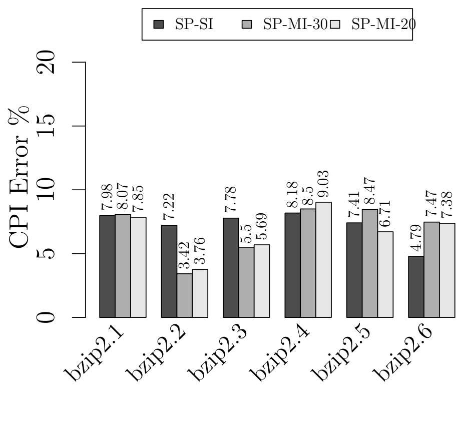
4.3.4 Phase Sharing Analysis

In the previous sections, we have shown the impact that shared phases have on the number of SimPoints and on the precision of the simulation results. Those results suggest that inputs share phases and that we can find a good phase characterization of the intervals from multiple inputs of a program. In this section, we characterize how those phases are shared among inputs, *i.e.*, which phases are present in each input and their contribution to each input. To do so, we first exemplify how sharing takes place using the `astar` benchmark.

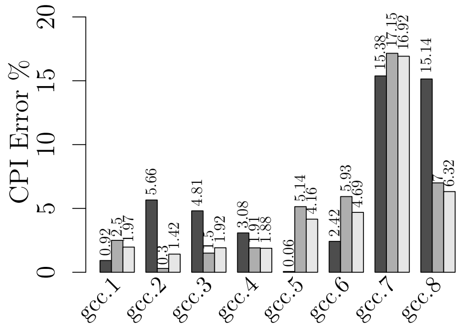
Figure 4.6 shows how the weights of the SimPoints differ for distinct inputs, in percentages of the execution time. The top bar represents the phases of the first input and their weights. Only six phases play relevant role for input 1 execution. The bottom bar shows the second input. Although the light green phase (rightmost) covers more than half of the program execution, this input takes the program through more distinct phases than input 1. Recalling Figure 4.4, by sharing several SimPoints, we could reduce their total number (from 27 to 18, with $maxK = 30$) but still keep a similar precision.



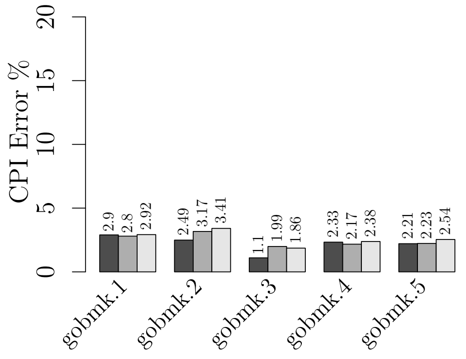
(a) perlbench



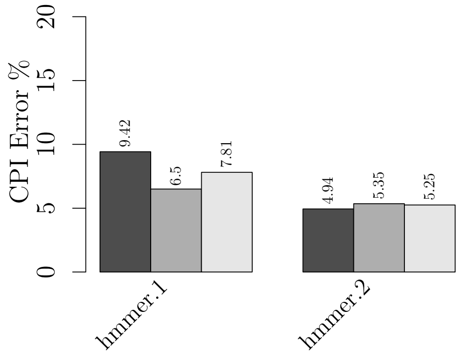
(b) bzip2



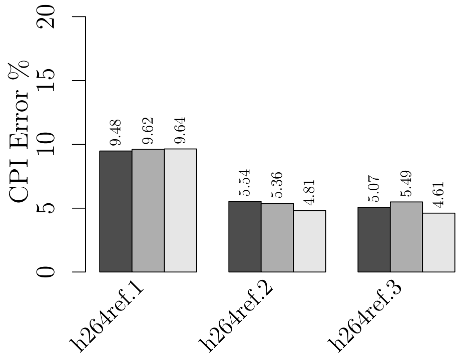
(c) gcc



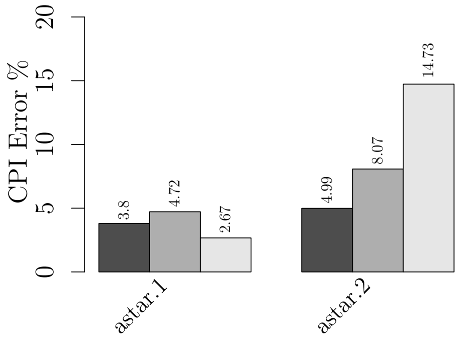
(d) gobmk



(e) hmmer



(f) h264ref



(g) astar

Figure 4.5: CPI evaluation for single input and multi-input ($maxK = 30$ and $maxK = 20$)

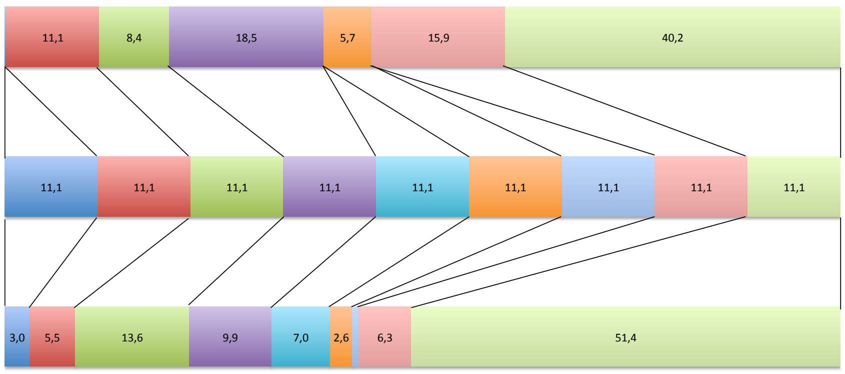
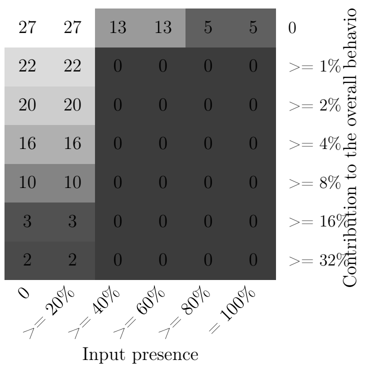


Figure 4.6: Distribution of SimPoints weigths for two inputs of the `astar` benchmark: input 1 on top and input 2 on bottom. Smaller weights omitted due to space restrictions. The middle bar represents the collection of all simpoints of all inputs

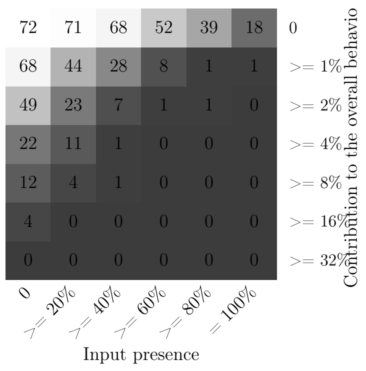
To quantify phase sharing, we characterized the number of phases and the percentage (in 20% increments) of the inputs that actually use them. Figure 4.7 shows our experimental results. For instance, `gcc` has 32 phases that are present in all inputs, 49 phases that are present in at least 80%, 68 phases present in at least 60%, and so on. The remainder of the results can be seen on the first line of each subfigure. Figure 4.7 also shows the contribution (weight) of those shared phases to the overall behavior of the inputs. These results are shown in lines (note the log axis). For instance, even though `gcc`'s inputs share 32 phases, only one of those phases covers every input with a contribution of at least 4% (4^{th} line), and none has a contribution of at least 32%.

The heatmap shown in Figure 4.7 also allows us to quickly observe the difference in phase sharing behavior between the benchmarks. For instance, `astar` has, proportionally, a larger number of phases shared among the multiple program inputs which can be seen by the smaller number of dark cells on its heatmap. In particular, `astar` has one phase that is present in all inputs with a contribution of more than 32%, which is not the case for any other evaluated benchmark.

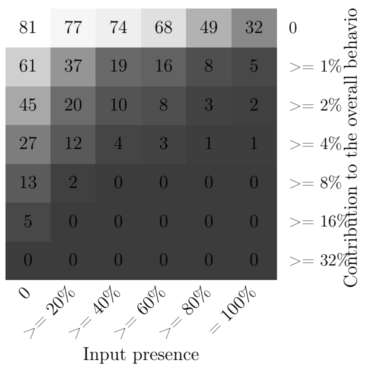
In this chapter, we showed that by taking into account similarities in the program behavior among different inputs, we can further reduce the time it takes to get simulation results of entire benchmarks. Specifically for SPECint 2006, we showed that the number of SimPoints (which is directly proportional to the simulation time) can be reduced by an average of 32% while losing only 0.06% of the accuracy when compared to the original technique. Further decreasing the accuracy by 0.5%, we observed the simulation time is reduced by an average of 66%



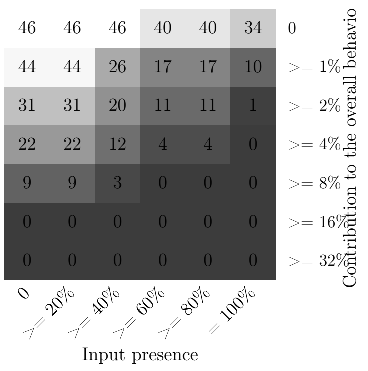
(a) perlbench



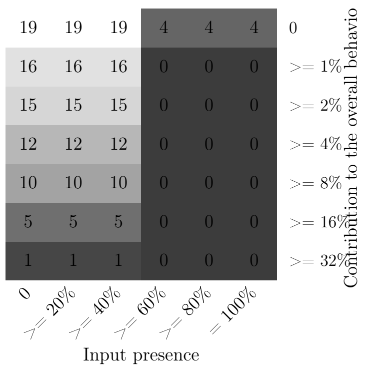
(b) bzip2



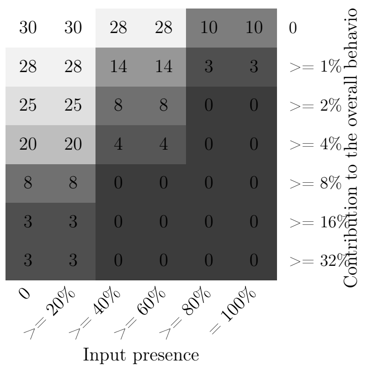
(c) gcc



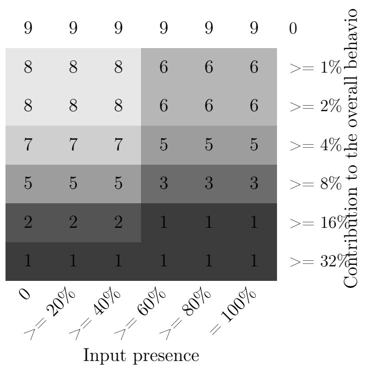
(d) gobmk



(e) hmmer



(f) h264ref



(g) astar

Figure 4.7: Heatmap showing how many phases are present and their coverage

Chapter 5

Conclusion and future work

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 supports multiple architectures (x86-64, x86, Arm, Aarch64) and runs on modern Linux systems. 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.

In the context of DBI frameworks, we also explored and studied the usage of DBI tools in other fields of computer science. As a joint effort with another graduate student from the Computer Systems Laboratory (LSC) at Unicamp, Rafael Mendonça Soares, we studied techniques to predict the performance of a given computer architecture when executing a particular workload based on information collected with the help of DBI tools. More specifically, we extended the SimPoint methodology, which originally was intended to find redundancy on program behavior of a single program on a single input, to explore redundancy on program behavior for multiple inputs on the same program at once.

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.

5.1 Future Work

As future work, we intend to:

- Continue increasing the support for more Pin API functions.
- Look for bottlenecks in DrPin performance and study alternative paths for the implementation of resource-heavy API functions.
- Completely support Zsim, without any instability.

5.2 Publications

We published our work on the extension of the Simpoints Technique in the proceedings of the Brazilian Symposium on High-Performance Computational Systems (WSCAD-2018) [39], and our work on DrPin will be published in the proceedings of WSCAD-2020.

Bibliography

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[2] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[3] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 265–275. IEEE, 2003.

[4] Prashanth P Bungale and Chi-Keung Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 137–147. ACM, 2007.

[5] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12. ACM, 2011.

[6] Trevor E Carlson, Wim Heirmant, and Lieven Eeckhout. 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, 2011.

[7] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[8] Michael J Eager et al. Introduction to the dwarf debugging format. *Group*, 2007.

[9] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 2–12, Oct 2005.

[10] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[11] Lieven Eeckhout. *Computer architecture performance evaluation methods.* Morgan & Claypool Publishers, San Rafael, CA, USA, 2010.

[12] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[13] Maxiwell Garcia, Emilio Francesquini, Rodolfo Azevedo, and Sandro Rigo. Hybrid-verifier: A cross-platform verification framework for instruction set simulators. *IEEE Embedded Systems Letters*, 9(2):25–28, 2017.

[14] Greg Hamerly, Erez Perelman, and Brad Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):25–30, 2004.

[15] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[16] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[17] Canturk Isci and Margaret Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *HPCA*, volume 3, pages 121–132, 2006.

[18] Björn Karlsson. *Beyond the C++ standard library: an introduction to boost.* Pearson Education, 2005.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[21] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[22] Onur Mutlu and Thomas Moscibroda. 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, 2007.

[23] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 63–74. IEEE Computer Society, 2008.

[24] Arun A Nair and Lizy K John. Simulation points for spec cpu 2006. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 397–403. IEEE, 2008.

[25] Nicholas Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004.

[26] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[27] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 81–92. IEEE, 2004.

[28] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92. IEEE Computer Society, 2004.

[29] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[30] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[31] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

[32] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486. ACM, 2013.

[33] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[34] MXVMJ Sheldon and Ganesh Venkitachalam Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[35] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 3–14. IEEE, 2001.

[36] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, October 2002.

[37] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE micro*, 23(6):84–93, 2003.

[38] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[39] Rafael Soares, Luis Antonioli, Emilio Francesquini, and Rodolfo Azevedo. Phase detection and analysis among multiple program inputs. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 155–161. IEEE, 2018.

[40] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84–95. IEEE, 2003.

[41] Heng Yin and Dawn Song. Temu: Binary code analysis via whole-system layered annotative execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[42] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework. *ACM SIGPLAN Notices*, 50(7):147–160, 2015.

Appendix A

DrPin API

Function	PIN_Init
Description	Initialize DrPin system. Should be the first API function to be called.
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_AddInstrumentFunction
Description	Resgister an instrument function that will be called every time DrPin finds an instruction for the first time. The instrument function has the opportunity to add instrumentation before or after this instruction.
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddFiniFunction
Description	Register a function to be called before the application exits. This function cannot insert instrumentation.
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_StartProgram
Description	Should be called before the application starts executing.
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_InsertCall
Description	Add instructions to call an analysis function before a given instruction
x86/x86-64	yes
ARM/AArch64	yes, but currently is not as optimized as the x86 version. It also supports fewer options

Function	INS_Address
Description	Returns the address of the instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Disassemble
Description	Disassembles the instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	REG_FullRegName
Description	Returns the name of the full name of the register. For example, on IA-32, if input is REG_AL, the function will return REG_EAX. If reg is a full register, the function returns it unchanged.
x86/x86-64	yes
ARM/AArch64	Partially

Function	REG_StringShort
Description	Converts register into a printable string
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandCount
Description	Returns the number of operands for the instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandRead
Description	Returns if the operand is a source
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandWritten
Description	Returns if the operand is a destination
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandIsMemory
Description	Returns if the operand is a memory reference
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandIsReg
Description	Returns if the operand is a register
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandReg
Description	Returns the register name for the operand
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandMemoryBaseReg
Description	Returns the register used as base register in memory operand, or REG_INVALID().
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_OperandMemoryIndexReg
Description	Returns the register used as index register in memory operand
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_LockPrefix
Description	Returns if the instruction has a lock prefix
x86/x86-64	yes
ARM/AArch64	No

Function	INS_OperandWidth
Description	Returns the operand width in bits
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Category
Description	Returns the category of the instruction. The category is a high level grouping of instructions. For exemple, XED_CATEGORY_COND_BR is the category of all instructions that are conditional branches
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Opcode
Description	Returns the opcode of the instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsAtomicUpdate
Description	Returns if the instruction may do an atomic update of memory
x86/x86-64	yes
ARM/AArch64	no

Function	INS_OperandIsImmediate
Description	Returns if the operand is an immediate
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsDirectBranch
Description	Returns if instruction is a direct branch
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_NumIns
Description	Returns the number of instructions within the basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_SpawnInternalThread
Description	Creates a new thread for the tool in the current process
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Next
Description	Returns the instruction that follows the current one in the current basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Valid
Description	Returns if the instruction is valid
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_Size
Description	Returns the number of instructions within the given basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_InsHead
Description	Returns the first instruction of the basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_Size
Description	Returns the size of the instruction in bytes
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_Address
Description	Returns the address of the first instruction in the given basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_ThreadId
Description	Returns the ID of the current thread
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_GetSyscallNumber
Description	Returns the number of the system call currently being executed. This function should only be called inside a SYSCALL_ENTRY_CALLBACK
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_SetSyscallArgument
Description	Sets the given value for the argument of the system call currently being executed. This function should only be called inside a SYSCALL_ENTRY_CALLBACK
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_GetSyscallArgument
Description	Gets the value for the argument of the system call currently being executed. This function should only be called inside a SYSCALL_ENTRY_CALLBACK
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_SetContextReg
Description	Set the given value for the register in the specified context
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_SetSyscallNumber
Description	Sets (overrides) the system call number for currently being executed. This function should only be called inside a SYSCALL_ENTRY_CALLBACK
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_GetContextReg
Description	Gets the value for the given register in the specified context
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_SafeCopy
Description	Copies a specified amount of bytes from a source memory address to a destination memory address
x86/x86-64	yes
ARM/AArch64	yes

Function	TRACE_BblHead
Description	Returns the first basic block (BBL) of the given trace
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_Valid
Description	Returns if the basic block is valid
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_InsertCall
Description	Add instructions to call an analysis function before a given basic block
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_Next
Description	Returns the following basic block relative to the current basic block within a trace
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_GetPid
Description	Returns the current process ID
x86/x86-64	yes
ARM/AArch64	yes

Function	GetVmLock
Description	Get Pin VM lock. Should be used to avoid race codition when the tool spawns multiple internal threads.
x86/x86-64	yes
ARM/AArch64	yes

Function	ReleaseVmLock
Description	Relead Pin VM lock. Should be used to avoid race condition when the tool spawns multiple internal threads
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_InitSymbols
Description	Initializes symbols table. Should be called before doing any symbol lookup. Must also be called before PIN_StartProgram
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddInternalExceptionHandler
Description	Registers a function that is called upon receipt of any unhandled internal exception in DrPin or the tool
x86/x86-64	yes
ARM/AArch64	yes

Function	TRACE_AddInstrumentFunction
Description	Resgister an instrument function that will be called every time DrPin constructs a new trace.
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddThreadStartFunction
Description	Register a function that will be called every time a new thread starts executing in the application. This function is also called for the main thread of the application
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddThreadFiniFunction
Description	Registers a function that will be called every time thread finishes in the application. This function is also called for the main thread of the application
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddSyscallEntryFunction
Description	Registers a function to be called immediately before the execution of a system call
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddSyscallExitFunction
Description	Register a function to be called immediately after the execution of a system call
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddContextChangeFunction
Description	Registers a function that will be called immediately before the application changes context due to an unix signal event
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddFollowChildProcessFunction
Description	Registers a function to be called before a child process starts to execute
x86/x86-64	yes
ARM/AArch64	yes

Function	PIN_AddForkFunction
Description	Registers a function to be called when the application forks a new process
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsMemoryRead
Description	Returns if the instruction reads memory
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsMemoryWrite
Description	Returns if the instruction writes memory
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsPredicated
Description	Returns if the instruction is predicated
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_HasMemoryRead2
Description	Returns if the instruction has 2 memory operands
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsXchg
Description	Returns if the instruction is an xchg
x86/x86-64	yes
ARM/AArch64	no

Function	INS_IsRDTSC
Description	Returns if the instruction is rdtsc or rdtscp
x86/x86-64	yes
ARM/AArch64	no

Function	INS_IsCJmp
Description	Returns if the instruction is a conditional jump instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsBranch
Description	Returns if the instruction is a branch instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsCall
Description	Returns if the instruction is a Call instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsRet
Description	Returns if the instruction is a Return instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	SEC_Valid
Description	Returns if the section is valid
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_IsDirectBranchOrCall
Description	Returns if the instruction is a direct branch or a call
x86/x86-64	yes
ARM/AArch64	yes

Function	INS_DirectBranchOrCallTargetAddress
Description	Returns the target address of the direct branch or the call instruction
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_AddInstrumentFunction
Description	Resgister an instrument function that will be called every time DrPin finds a new image.
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_AddUnloadFunction
Description	Registers a callback function to be called when this image is unloaded from memory. It is not possible to add any instrumentation inside this function.
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_Close
Description	Closes the open image
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_FindByAddress
Description	Returns the image that contains that address
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_HighAddress
Description	Returns the highest address of the image
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_IsMainExecutable
Description	Returns if the image is the main executable
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_LowAddress
Description	Returns the lowest address of the image
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_Name
Description	Returns the image name
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_Open
Description	Open the image. The image should be opened before one can traverse it staticaly
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_SecHead
Description	Returns the first section of the image
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_Type
Description	Return the image type. Possible values are: IMG_TYPE_STATIC, IMG_TYPE_SHARED, IMG_TYPE_SHAREDLIB, IMG_TYPE_DYNAMIC_CODE
x86/x86-64	yes
ARM/AArch64	yes

Function	IMG_Valid
Description	Returns if the image object is valid
x86/x86-64	yes
ARM/AArch64	yes

Function	BBL_InsTail
Description	Returns the last instruction of the basic block
x86/x86-64	yes
ARM/AArch64	yes