Individual Diploma Thesis

Dynamic Trace Segmentation to Maximize ILP Utilization in Superscalar CPUs

Giorgos Hadjipavlou

University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

May 2025

University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

Dynamic Trace Segmentation to Maximize ILP Utilization in Superscalar CPUs

Giorgos Hadjipavlou

Supervisor
Dr. Sazeides Yanos

The Individual Diploma Thesis was submitted in partial fulfillment of the requirements for the acquisition of the degree in Computer Science of the Department of Computer Science at the University of Cyprus.

May 2025

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Dr. Yanos Sazeides for his invaluable guidance, support, and mentorship throughout the course of this thesis. His deep expertise, insightful feedback, and encouragement have been invaluable in shaping both the direction and the quality of my work. It has been a privilege to work under his supervision and to learn from his experience.

Furthermore, I would also like to thank my colleagues and classmates at the university, whose discussions, feedback and support have been incredibly helpful as well as motivating throughout the whole year of my research.

Finally, I am deeply grateful to my family and friends for their continuous support, patience, and understanding. I could not have done it without them.

Abstract

Modern superscalar CPUs achieve high performance by executing multiple instructions per clock cycle, relying on instruction-level parallelism (ILP). However, a major barrier to fully exploiting ILP lies in control flow instructions, which introduce ambiguity into the instruction stream. When a branch is encountered, the processor cannot immediately determine the next instruction to fetch until the branch is resolved. This uncertainty forces the pipeline to stall, awaiting resolution, and results in executing less Instructions Per Cycle (IPC) than the architecture is capable of. While branch prediction mechanisms are employed to mitigate these delays, the increasing number of branches in complex programs leads to greater prediction overhead and higher misprediction.

This thesis addresses the instruction supply bottleneck through a trace-based approach. The proposed method generates traces, linear instruction sequences that span multiple basic blocks, using configurable segmentation criteria. These criteria include: a maximum number of instructions, a maximum number of control flow instructions (CFIs), and encountering specific control flow instruction which based on the configuration can terminate the trace.

Traces are created dynamically during execution. Afterwards each unique trace is identified using a starting address, length, CFI count, occurrence frequency, and termination cause. This metadata enables evaluation of how different segmentation strategies affect trace structure and potential instruction throughput. Lastly, by analyzing these it is determined what configurations produced more desirable outcomes.

The results demonstrate that carefully configured trace segmentation can mitigate the disruptive impact of frequent branching, enhance instruction predictability, and increase the efficiency of instruction supply, ultimately enabling more effective ILP exploitation in superscalar CPUs and achieving higher IPC as well as faster execution times.

Contents

1 Chapter 1: Introduction

As CPU architecture becomes more sophisticated with greater execution capabilities it is imperative that microarchitectural components evolve in tandem as to not bottleneck the overall performance of the CPU.

1.1 Motivation

A Level 1 and Level 2 top-down analysis was performed for benchmarks from the SPEC 2017 Suite. Table 1.1 shows the CPU used to run the benchmarks.

Component	**Description**
Architecture	x86_64
CPU Modes	32-bit, 64-bit
Address Sizes	46 bits physical, 48 bits virtual
Byte Order	Little Endian
Total CPU Threads	20
Online CPUs	0–19
Vendor ID	GenuineIntel
Model Name	12th Gen Intel(R) Core(TM) i7-12700
CPU Family	6
Model	151
Stepping	2
Cores per Socket	12
Threads per Core	2
Sockets	1
CPU Min Frequency	800 MHz
CPU Max Frequency	4900 MHz
BogoMIPS	4224.00
L1 Data Cache	512 KiB (12 instances)
L1 Instruction Cache	512 KiB (12 instances)
L2 Cache	12 MiB (9 instances)
L3 Cache	25 MiB (1 instance)

Table 1.1: CPU Information

Top-down analysis is a method to categorize where performance bottlenecks occur during execution. For Level Top-Down analysis the CPU pipeline behavior is divided into four broad parts, which then more layers expand upon. It is used to pinpoint exactly the reason the CPU is not being fully utilized.
Level 1 Top-Down Microarchitectural Analysis [1] categorizes processor pipeline slots into four main groups:

Retiring: Slots where useful micro-operations (uOps) are executed and retired; indicates effective use of the pipeline.

Bad Speculation: Slots were wasted due to incorrect speculative execution, such as branch mispredictions or pipeline clears.

Frontend Bound: Stalls caused by the frontend failing to deliver uOps to the backend (e.g., due to instruction cache misses or decode inefficiencies).

Backend Bound: Stalls occurring when the backend cannot process uOps despite a ready frontend, often due to resource contention or memory latency.

Figure 1.1 shows the results of Level 1 Top-Down Analysis along with a black line representing Instructions Per Cycle (IPC).

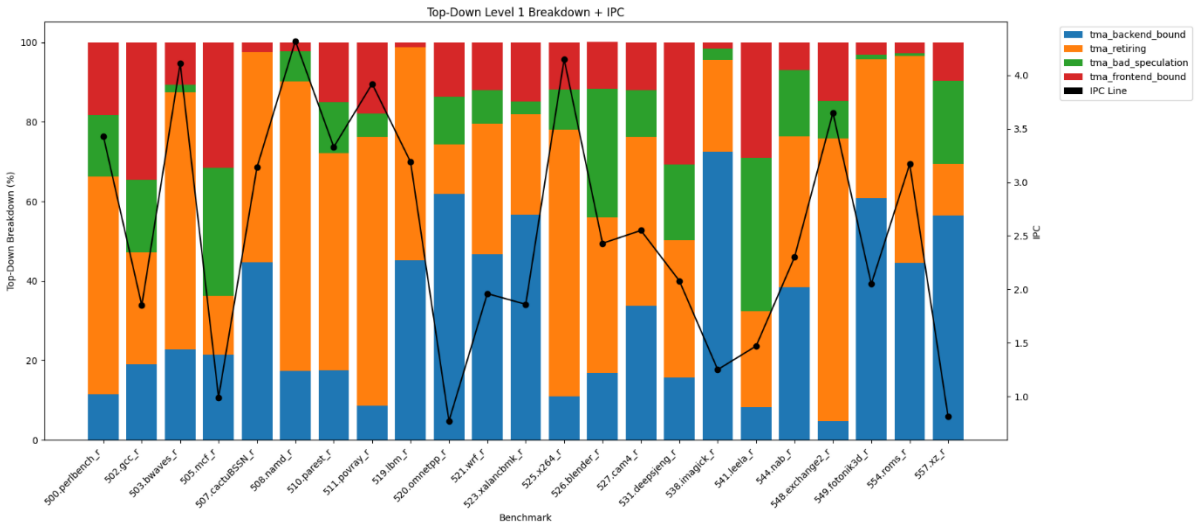

Figure 1.1: Level 1 Top-Down Analysis

This thesis focuses on bad speculation which is the root of the problem it aims to improve upon. Bad speculation is represented by the colour green. Although it is not the only contributor to low IPC, it is a critical one, particularly in modern superscalar out-of-order processors where speculative execution is aggressive and frequent. It can be observed from Figure 1.1 that high bad speculation leads to a considerable decrease in IPC which is represented by a black line. Even further analysis at level 2 branch miss-predicts show a clearer picture of this phenomenon. Figure 1.2 shows the results of Level 2 Top-Down Analysis

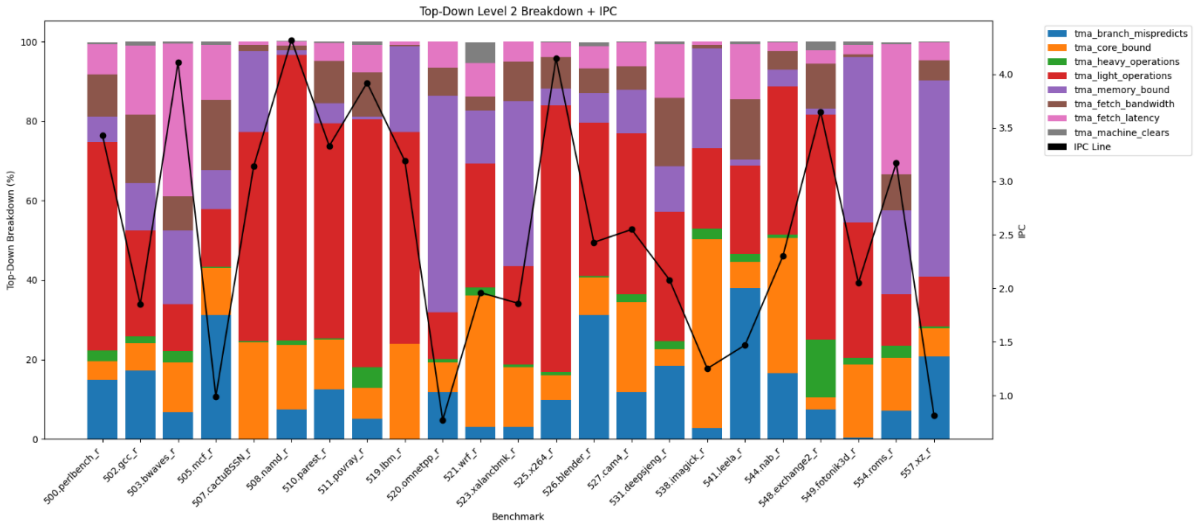

Figure 1.2: Level 2 Top-Down Analysis

In level 2, it can be observed that branch miss-prediction which is represented by the colour blue and is at the bottom of the figure, has a negative correlation with Instructions executed per Cycle (IPC), which again is represented by a black line.

Figure 1.3 is a clear summary of both Figures 1.1 and 1.2 including only the focus of the analysis, bad speculation and branch miss-predicts.

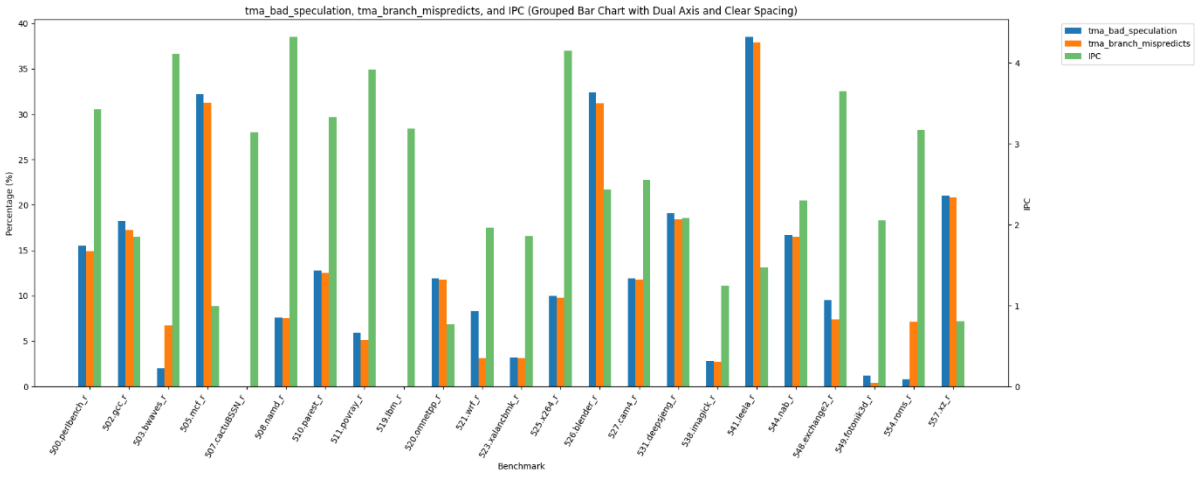

Figure 1.3: Level 1&2 Top-Down Analysis

It should be noted that there are other factors that affect IPC, not only bad speculation but it is a primary factor. A common pattern that appears is either IPC is high, represented in green, or bad speculation and branch miss-predicts are high, represented by blue and orange respectively.

1.2 Problem Statement

Modern CPU architecture continues to evolve and support incredible throughput capabilities. Utilizing them fully has proven to be a major challenge since, particularly in superscalar CPUs who are designed with instruction-level parallelism (ILP) as the focus and are able to execute multiple instructions at the same time. The biggest obstacle yet has not actually been increasing the processing power but being able to fully utilize a superscalar CPU to its fullest potential. In a perfect environment, one would be able to simply predict what instructions will be executed in the future and simply execute them all in parallel but with any problem that requires any sort of logic this would not be feasible. The problem of always being able to predict the upcoming instructions is control flow instructions (CFIs), which alter control flow based on the current state and are not consistent.

1.3 Contributions

- **Design of a Configurable Trace Segmentation Policy:**
  A rule-based trace segmentation policy was developed, allowing configurable termination conditions based on maximum instruction count, control-flow instruction count, and specific terminating control-flow instruction types.

- **Implementation of a Custom Pin Tool for Trace Collection:**
  A custom Intel Pin-based tool was developed to efficiently extract traces, incorporating the segmentation policy with support for large-scale trace generation.

- **Optimizations for Performance Increase:**
  Techniques such as minimal memory allocation, selective instrumentation, and an efficient algorithm were applied to improve runtime efficiency and reduce overhead during trace collection.

- **Evaluation of Trace Segmentation Configurations:**
  A comprehensive analysis was performed on multiple segmentation configurations to understand their impact on trace quality, length, frequency of execution, and overall trace set size.

- **Insight into Trade-offs in Trace Segmentation:**
  The results provide insight into how different segmentation parameters affect the trace set and the trade-offs of each configuration based on what system this each trace segmentation configuration will be used.

1.4 Outline

Chapter 2 will go into technical background regarding what is ILP, how OoO superscalar execution exploits ILP and the challenges it faces due to unpredictable control flow. Chapter 3 will go over what trace-based execution and systems surrounding execution revolving traces. Chapter 4 will go into related work. Chapter 5 will explain the methodology of what different configurations were tested. Chapter 6 shows the actual implementation of how the data was collected and processed. Chapter 7 will include my results along with evaluations of the findings. Chapter 8 finally concludes the thesis and sets up for future work.

2 Chapter 2: Technical Background

In this chapter will explore what Instruction Level Parallelism (ILP), mechanism used to exploit ILP and the challenges that control flow brings to ILP.

2.1 Instruction Level Parallelism (ILP)

Instruction Level Parallelism is a method to exploit parallelism and execute multiple instructions simultaneously using a single thread and a single core. By executing instructions in parallel it is possible to execute more instructions per cycle (IPC) which leads to a substantial decrease in execution times.

Instruction Level parallelism is one form of parallelism amongst others, such as thread level parallelism, data level parallelism and task level parallelism. The goal of parallelism is to execute multiple instructions in tangent to ultimately utilize more resources to reduce execution time. The distinction regarding Instruction Level Parallelism and what makes it unique among the other types of parallelism is that ILP is that it is extracting parallelism from a single instruction stream, whereas other forms operate across multiple threads, tasks, or data. ILP happens inside a single core, while others often happen across cores or units. ILP occurs automatically and does not require additional resources and implementation, in contrast to TLP for example that requires multiple cores and parallel coding.

ILP extracts parallelism from a single, sequential instruction stream, executing independent instructions in parallel while preserving the appearance of in-order execution. While the program is written as a linear sequence of instructions, the CPU dynamically analyzes the stream to identify which instructions can be executed simultaneously. Essentially mimicking what a sequential execution would be, which is what makes it so fundamentally important. All executions can benefit from instruction level parallelism even when the code is written with no parallelism in mind, ILP can still reduce execution times with no additional effort. It also makes sure that no processing power gets wasted and all resources are being always fully utilized during execution.

2.2 Techniques for Exploiting Instruction-Level Parallelism

To exploit ILP in hardware, modern processors use several architectural techniques that allow multiple instructions to be executed simultaneously or. These techniques include pipelining, out-of-order execution (OoO) and superscalar execution.

2.2.1 CPU Pipeline

Figure 2.1 shows a basic CPU Pipeline which represents the stages in which an instruction goes through. Starting from when the instruction first enters the CPU up to finally writing in memory.

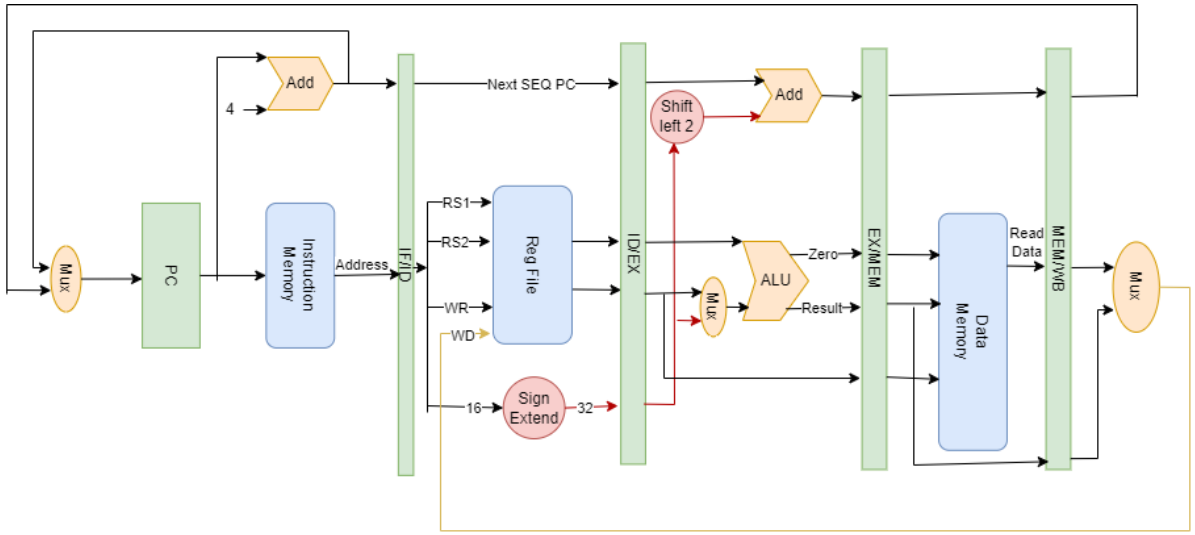

Figure 2.1: CPU Pipeline

The 5 main stages are

Instruction Fetch (IF): The CPU retrieves instruction from program memory

Instruction Decode (ID): The instruction is interpreted, and the required operands are prepared

Execute (EX): The instruction's operations are performed

Memory Access (MEM): The instruction accesses the memory to read and write data

Write Back (WB): The result is written in the CPU's register file

Figure 2.2 is an example of a single instruction going through every phase of the pipeline
The number represents Clock Cycles.
Instruction 1 goes through each stage of the pipeline and each stage of the pipeline takes 1 clock cycle to complete.

	1	2	3	4	5
INS 1	IF	ID	EX	MEM	WB

Figure 2.2: Single Instruction Pipeline

Instruction 1 enters the pipeline, the first clock cycle and is the Instruction Fetch (IF) stage and finally exists the pipeline in the fifth clock cycle after Write Back (WB)

This in theory would have an IPC of 0.2 since it takes 5 cycles for 1 instruction, essentially equivalent to 0.2 instructions per cycle.

With this simplistic design though the pipeline is not fully utilized.

In modern CPUs, while an instruction is in the pipeline another instruction can enter, given that they are in different stages of pipeline. This allows the CPU to process multiple instructions concurrently, one at each stage. If theoretically the whole pipeline is being used with distinct instructions it can execute 5 instructions simultaneously since each instruction would be at different stage. One instruction for each stage would mean an IPC of 1 since every clock cycle one instruction would enter and one instruction would leave the pipeline.

Figure 2.3 is an example of 2 instructions at different phases of the pipeline executing in parallel. Instruction 1 enters first then is immediately followed by Instruction 2 in the next clock cycle.

	1	2	3	4	5	6
INS 1	IF	ID	EX	MEM	WB	
INS 2		IF	ID	EX	MEM	WB

Figure 2.3: CPU Pipelining

2.2.2 Out-of-Order Execution (OoO)

Since with the use Pipelining, it is possible to execute multiple instructions at the same time but there is another matter at hand. This approach seems promising but in the scenario that an instruction stalls at a specific point in the pipeline then every instruction would have to wait until said instruction enters the next stage. If, for example, an instruction stalls at execution since there exists a data dependency or any other hazard the whole pipeline would come to a halt and wait for that instruction to continue in the pipeline. This scenario is exactly what Out-of-Order (OoO) Execution combats. Out-of-order execution allows for instructions to continue into the next stage of the pipeline, if it is available, regardless of when they entered the pipeline, or their order in the program. In this scenario where an instruction stalled due to a data dependency and cannot be executed other instructions can enter the execution stage. As soon as their operands are ready, they can be executed, rather than wait to strictly adhere to the order they appear in the program or the pipeline.

Figure 2.4 is an example of an instruction being executed out of order due to the previous instruction stalling before execution. Since instruction's 2 operands were ready before instruction's 1 it went ahead in the execution phase first and even exited the pipeline first although it entered second.

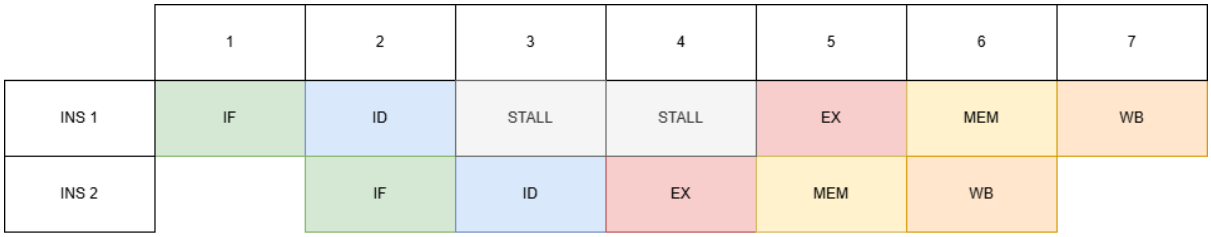

Figure 2.4: OoO Execution

2.2.3 Superscalar Execution

A superscalar processor is a CPU that issues multiple instructions per clock cycle, superscalar execution. Its key feature is that it contains multiple execution pipelines. A scalar CPU is only able to execute one instruction per clock cycle but for example a 4-wide superscalar can issue 4 instructions simultaneously essentially achieving an IPC of 4 in perfect conditions.

Figure 2.5 is an example of superscalar execution where two instructions enter a 2-wide superscalar processor in 2 distinct pipelines.

	1	2	3	4	5
INS 1	IF	ID	EX	MEM	WB
INS 2	IF	ID	EX	MEM	WB

Figure 2.5: Superscalar Execution

Instructions 1 and 2 enter and leave the CPU in the same clock cycle. They go through each stage simultaneously in parallel.

2.2.4 ILP Techniques in Action

Figure 2.6 depicts a potential state of a modern CPU that utilizes ILP. This is a superscalar CPU with the ability to issue 2 instructions at a time. It has 2 distinct pipelines and in those distinct pipelines there can be instructions in each phase. In addition, execution can happen out-of-order. All techniques used to exploit ILP can be seen in.

Instructions 1 and 3 share the same pipeline and instructions 2 and 4 share the other pipeline.

	1	2	3	4	5	6	7
INS 1	IF	ID	EX	MEM	WB		
INS 2	IF	ID	STALL	STALL	EX	MEM	WB
INS 3		IF	ID	EX	MEM	WB	
INS 4		IF	ID	EX	MEM	WB	

Figure 2.6: Example of ILP

It can be observed from Figure 2.6 that since instructions 3 and 4 do not share a pipeline they can be executed perfectly in parallel and go through each stage in unison. Additionally, instruction 2 stalls but due to the use of out-of-order execution even though instruction 4 shares the same pipeline as 2, and even though it entered the pipeline before it, it can proceed without having to stall wait for instruction 2 to finish.

2.3 Control Flow

Using all the techniques previously mentioned a very high IPC can be achieved. If a steady stream of instruction is kept it is possible to fully utilize all the resources available a superscalar CPU has to offer. This is all great in theory but achieving a steady stream of instructions is much harder in practice.

The reason this is the case is that control flow which is the order in which a program's instructions are executed is not as simple as sequentially executing instructions in program order. If logic is introduced the next instruction that needs to be executed might not be the next instruction is program order but a completely different one instead. This is what is called a control flow change. There are several instructions that control and affect control flow. These instructions are called control flow instructions.

2.3.1 Control Flow Instructions

Control Flow Instructions (CFIs) are instructions that alter the sequential order of execution, this essentially means that after a control flow instruction the next instruction might not be the next instruction in program order but instead a completely different one.

The types of control flow instructions that will be investigated can be seen in Table 2.1 (the examples are x86 instructions):

Name	**Description**	**Example**
Return	Returns from a function caller	RET
Direct Call	Calls a function with a known address	CALL 0x401000
Indirect Call	Calls a function via a register or memory location	CALL EAX
Direct Conditional Jump	Jumps to a fixed target address if a condition is met	JE 0x401020
Direct Unconditional Jump	Always jumps to a fixed target address unconditionally	JMP 0x401030

Indirect Conditional Jump	Jumps to a computed address if a condition is met	JZ DWORD PTR [EAX]
Indirect Unconditional Jump	Always jumps to a computed address unconditional	JMP EAX

Table 2.1: Control Flow Instructions

With each control flow instruction, the control flow branches into distinct paths which is why they are also called branches. Branches introduce uncertainty in the control flow. What this essentially means is that after a branch there is no one deterministic path. Branches have conditions and if those conditions are met that means then the processor must jump to a different instruction address specified by the branch. Otherwise, the execution continues with the instruction immediately following the branch in program order. When a branch is taken that means the conditions of the jump are met and so the control flow will not follow the instructions preceding the branch but instead the target address of the branch. Even when a control flow change is predictable, meaning it is an unconditional jump, its target address might still not be hardcoded and instead be from a register or memory. This creates another layer of uncertainty to what might seem like an easy process but if the branch is not resolved the program order cannot be determined, since there is a need to know both the direction, taken not taken, but also need to know the target address.

2.3.2 Branch prediction

During the execution of a program that includes some sort of logic it is inevitable that a branch will occur which means in order to keep the pipeline full a prediction needs to be made so that the pipeline does not stall and wait until the branch is finally resolved. This is where a CPU component called a branch predictor comes into conversation. The job of a branch predictor is to make an educated guess as to what the next instruction preceding a branch is going to be. This is called Speculative Execution.

A branch prediction occurs at the fetch stage of the pipeline when a branch instruction first enters the pipeline. Here the CPU determines that a branch has occurred and using the branch predictors output finds what instruction it will fetch after the branch before resolution.

Branch Predictors attempt to find the next instructions by storing data regarding previous encounters of said branch as well as pattern recognition. For example, by storing data regarding the branch history in a branch history table, branch target buffers or even the return address stack. Using this data the branch predictor can detect patterns regarding the outcomes of branches. Using data based on previous times the branch occurred and combining it with data regarding what happened before the branch the predictor estimates what the next instruction will be and once the branch is finally resolved adjusts based on the outcome. If a wrong prediction occurs, then every instruction executed in the wrong path will be flushed. A branch predictor will adjust during execution based on more recent patterns it will detect. The nature of a branch can change during execution leading to branch predictors having a recency bias as to adjust their predictions.

2.3.3 Limitation of Branch Predictors

Branch predictors are designed in a way to predict one branch at a time which becomes a bottleneck when more and more instructions are required. Architectures that are capable of high IPCs mean that they require a lot of instructions which inherently result in the need for more frequent branch prediction. Predicting multiple branches so frequently can prove to be challenging for branch predictors due to the ambiguity each distinct branch brings in the control flow as well as the cost of predicting and adjusting after a branch has been resolved. Also using information like branch history is challenging because there the information regarding what happened in previous branches might not be available during prediction if for example, the previous branches have yet to be resolved that means their outcome is still speculative and not definitive. There is a need to quickly and efficiently predict multiple branches at a time to keep a steady instruction stream throughout execution. Branch Predictors also need time to “warm up” at first since the predictor is still learning the branch behavior of the program and cannot make accurate predictions.

3 Trace-Based Execution

Trace-Based Execution is one way to combat the shortcomings of branch predictors by replacing them with traces which can span multiple basic blocks and instead use a trace predictor.

3.1 Basic Blocks

A basic block is a straight-line sequence of instructions with no branches except the last instruction. They have a single entrance meaning control can only enter at the first instruction and a single exit meaning that control always leaves at the last instruction. There are no internal control flow instructions inside a basic block, there can only exist a control flow instruction as the last instruction of a basic block. What this essentially means is the fact that if the first instruction of a basic block is encountered then all the instructions in the basic block will be executed in sequence.

3.2 Traces

A trace is a sequence of instruction that can span multiple of basic blocks and so may include multiple control flow instructions. Traces are created at runtime by grouping together instructions that occur sequentially. The reason traces are created is instead of predicting one branch at time is that by using a trace predictor it is possible to essentially predict multiple consecutive branches at once. Now instead of predicting every single branch it will only predict every single trace. At runtime traces are created dynamically and stored in memory. Afterwards when a prediction is made that a trace will occur the whole trace enters the instruction stream and no other prediction occurs mid trace, it is assumed that every branch in the trace will take the same direction that is in the trace. Since the only prediction that occurs is between traces the goal is that the branches inside the trace are consistent and almost always follow the exact same path. Traces are creating using a trace segmentation policy which determines during execution whether a certain instruction will terminate the current trace and start or new one or be included in it. If an internal branch inside the trace follows a different direction when resolved then a flush occurs, it is treated the same as a branch miss-prediction.

3.3 Trace Predictor

A Trace Predictor works very similarly to a branch predictor but now instead of using branch history and previous occurrences of the branch it's trying to predict, it is using a trace history and previous occurrences of the trace it is trying to predict. The predictor also adjusts during execution to better recognize more recent patterns that occur in between traces. The concept is almost identical the only exception is that traces need to be created first. If a sequence of instructions has never occurred before and there is no trace stored that matches that sequence, then a new trace is created and stored.

3.4 Trace Cache

In a realistic real-life environment, it would be very difficult to store every single trace that has ever occurred during a program's runtime, which is why a cache-like storage system is used instead. A trace cache uses common functionalities that can be found in regular cache to store the traces. Common cache replacement policies such as Least Recently Used (LRU) and Least Frequently Used (LFU) are used in a trace cache as well. Since storing every single instruction included in a trace would require a lot of space, traces are instead stored using summarized trace metadata. Such metadata can be number of instructions, control flow events, termination reason, addresses, number of branches, number of control-flow-instructions as well as hashed signatures or even decoded micro-ops.

3.5 Trace Quality Metrics

There exists a multitude of ways to create traces with different trace segmentation policies but there is a more desirable outcome that shows that a trace segmentation policy is better than another. The criteria that define the quality of a trace set are as follows.

1. **Trace Length:** The reason traces are being used after all is to achieve higher ILP, which requires more instructions. To create a steady instruction stream and minimize the cost of fetching instructions and branch prediction, it is important to make longer traces that include a high number of instructions. Trace length is the reason traces are used, making it one of the most vital qualities for a trace set to have.

2. **Hotness and Execution Frequency:** The term “hot traces” refer to a collective of traces that occur frequently during execution. It is important that a small number of traces are executed far more frequently so that the parts of the code that make its core can be correctly identify. Creating traces but not being able to use them frequently defeats their purpose.

3. **Branch Consistency:** The branches that are included in a trace must be consistent to reduce the likelihood of pipeline flushes. To preserve consistency within the trace are often segmented at points of low predictability. Non-consistent traces will also have a lot of overlap between them, which is undesirable, as it would mean storing them would be even more challenging.

4. **Data Dependencies:** In order execute a trace using ILP there must not exist any data control dependencies between instructions. Data dependencies, although not a control hazard are important to consider when creating traces, or at least the order data dependent instructions enter the pipeline.

5. **Number of Unique Traces:** Having a lot of unique traces would require a lot of memory space to store them which even with very little traces can prove to be difficult, especially in long programs with a lot of instructions and branching paths. Even if a cache-like storage system is implemented, unique traces will still compete for space, possibly creating unnecessary overhead and bad replacements in cache.

4 Chapter 4: Related Work

Trace segmentation has been a foundational component in many trace-based microarchitectures, particularly those that aim to improve instruction fetch bandwidth and branch prediction accuracy. Early work by Rotenberg et al. [2] introduced fixed-length traces in a trace cache, where traces terminated at a maximum instruction count, basic block limit, or upon encountering an indirect control-flow instruction such as a return or indirect jump. Conditional branches and direct calls, however, were allowed within traces.

Subsequent work extended this model. Jacobson et al. [3] introduced a segmentation policy that allowed traces to contain up to six branch instructions, again terminating early at indirect branches to preserve trace identity. Rotenberg et al. [4] refined the trace cache architecture while maintaining similar segmentation policies based on trace length and branch type. Rosner et al. [5] explored longer atomic traces for high coverage and compared segmentation heuristics such as terminating at calls or backward branches to improve trace utility.

Unlike these studies, which tie segmentation policy to hardware implementation goals such as trace cache hit rate or fetch bandwidth, this thesis isolates the segmentation policy itself as the subject of evaluation. The implemented policy terminates traces based on three configurable parameters: a maximum instruction count, a maximum number of control-flow instructions, and specific types of control-flow instructions. By running experiments using this segmentation policy with multiple unique configurations the resulting trace sets are evaluated based on the format and how suitable they would be when implemented.

5 Chapter 5: Methodology

This chapter will go over experiments' methodology. What trace segmentation policy was implemented, on what workload as well as what different configurations were used.

5.1 Workloads

The SPEC 2017 benchmark suite was used as representative workload in which to test the different configurations of the trace segmentation policy. Tables 5.1 shows some general information about the benchmarks used:

Benchmark	Type	Language	KLOC	Application Area
500.perlbench_r	Integer	C	362	Perl interpreter
502.gcc_r	Integer	C	1304	GNU C compiler
503.bwaves_r	Floating Point	Fortran	1	Explosion modelling
505.mcf_r	Integer	C	3	Route planning
507.cactuBSSN_r	Floating Point	C++, C, Fortran	257	Physics: relativity
508.namd_r	Floating Point	C++	8	Molecular dynamics
510.parest_r	Floating Point	C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r	Floating Point	C++, C	170	Ray tracing
519.lbm_r	Floating Point	C	1	Fluid dynamics
520.omnetpp_r	Integer	C++	134	Discrete Event Simulation - computer network
521.wrf_r	Floating Point	Fortran, C	991	Weather forecasting

523.xalancbmk_r	Integer	C++	520	XML to HTML conversion via XSLT
525.x264_r	Integer	C	96	Video compression
526.blender_r	Floating Point	C++, C	1577	3D rendering and animation
527.cam4_r	Floating Point	Fortran, C	407	Atmosphere modeling
528.pop2_s	Floating Point	Fortran, C	338	Wide-scale ocean modeling (climate level)
531.deepsjeng_r	Integer	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
538.imagick_r	Floating Point	C	259	Image manipulation
541.leela_r	Integer	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
544.nab_r	Floating Point	C	24	Molecular dynamics
548.exchange2_r	Integer	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
549.fotonik3d_r	Floating Point	Fortran	14	Computational Electromagnetics

554.roms_r	Floating Point	C	210	Regional ocean modeling
557.xz_r	Integer	C	33	General data compression

Table 5.1: Benchmark Descriptions

Table 5.2 shows information regarding runtime metrics. The input for all the benchmarks was consistent throughout all runs.

Benchmark ID	CPU Core Cycles (B)	CPU Core Instructions (B)	CPU Core Branches (B)	CPU Core Branch Misses (M)	Time (s)	Branch Mispredicti Rate (%)	IPC (Instructions/Cycl	TMA Branch Mispredicts
500.perlbench_r	363B	1,243B	233B	2,049M	76.66	0.88	3.43	14.90
502.gcc_r	107B	198B	46B	963M	22.67	2.09	1.85	17.20
503.bwaves_r	358B	1,473B	173B	213M	75.38	0.12	4.11	6.70
505.mcf_r	27B	27B	7B	511M	5.82	7.10	0.99	31.30
507.cactuBSSN_	487B	1,533B	27B	16M	103.83	0.06	3.14	0
508.namd_r	602B	2,604B	42B	1,803M	127.25	4.27	4.32	7.50
510.parest_r	1,097B	3,659B	489B	8,377M	235.03	1.71	3.33	12.50
511.povray_r	900B	3,532B	562B	1,429M	193.53	0.25	3.92	5.10
519.lbm_r	520B	1,659B	15B	12M	111.13	0.08	3.19	0
520.omnetpp_r	1,397B	1,080B	230B	4,687M	299.44	2.03	0.77	11.80
521.wrf_r	1,908B	3,745B	368B	2,399M	403.16	0.65	1.96	3.10
523.xalancbmk_	701B	1,305B	385B	728M	149.03	0.19	1.86	3.10
525.x264_r	77B	319B	14B	449M	16.35	3.13	4.15	9.80
526.blender_r	763B	1,851B	307B	7,061M	161.37	2.29	2.43	31.20
527.cam4_r	1,134B	2,894B	356B	4,952M	239.63	1.39	2.55	11.80
531.deepsjeng_r	924B	1,920B	283B	8,574M	195.11	3.02	2.08	18.40
538.imagick_r	3,522B	4,414B	544B	2,303M	743.25	0.42	1.25	2.70
541.leela_r	1,498B	2,206B	362B	29,520M	316.32	8.13	1.47	37.90
544.nab_r	949B	2,186B	244B	6,148M	200.33	2.52	2.30	16.50
548.exchange2_	1,823B	6,650B	733B	5,408M	384.50	0.74	3.65	7.40
549.fotonik3d_r	1,074B	2,207B	45B	154M	226.79	0.34	2.05	0.40
554.roms_r	936B	2,965B	215B	423M	197.48	0.20	3.17	7.10
557.xz_r	488B	397B	53B	3,471M	103.62	6.49	0.81	20.80

Table 5.2: Benchmark Runtime Metrics

Due to time constraints the trace set will not be representative of all the instructions in all the benchmarks. Table 5.3 shows from which instruction up to which instruction the trace generation will begin and end. Instructions are represented in Billions.

benchmark	Starting Point (B)	Stopping Point (B)	Total (B)
500.perlbench_r	206	412	206
502.gcc_r	0	199	199
503.bwaves_r	336	672	336
505.mcf_r	0	27	27
507.cactuBSSN_r	0	1532	1532
508.namd_r	872	1745	872
510.parest_r	257	514	257
511.povray_r	213	426	213
519.lbm_r	0	659	659
520.omnetpp_r	147	295	147
521.wrf_r	315	630	315
523.xalancbmk_r	150	299	150
525.x264_r	0	319	319
526.blender_r	217	434	217
527.cam4_r	242	485	242
531.deepsjeng_r	242	484	242
538.imagick_r	322	643	322
541.leela_r	222	444	222
544.nab_r	303	606	303
548.exchange2_r	377	754	377
549.fotonik3d_r	0	2206	2206
554.roms_r	453	906	453
557.xz_r	0	397	397

Table 5.3: Start and Stop Points

5.2 Trace Segmentation Policy

The trace segmentation policy generates traces dynamically during execution by analyzing each instruction executed and then determining whether to terminate the trace using segmenting criteria. The segmenting criteria that result in termination of a trace are as follows:

1. The trace has reached the maximum number of instructions allowed.
2. The trace has reached the maximum number of control flow instructions allowed.
3. The last instruction encountered is a type of control flow instruction that terminates the trace.

When an instruction enters a trace all 3 criteria are checked and if at least one of them is true then the trace is terminated, and the next instruction will be part of a new trace.

See the following example. For this example, a trace terminates when encountering a call, when it reaches 8 instructions or 2 when it reaches control flow instructions.

1. mov rdi, rsp
2. call 0x7f63eba61880
Trace 1: 2 Instructions, 1 CFI, Terminated due to instruction type: Call
1. nop edx, edi
2. push rbp
3. mov rbp, rsp
4. push r15
5. mov r15, rdi
6. push r14
7. push r13
8. push r12
Trace 2: 8 Instructions, 0 CFI, Terminated due to max instructions
1. mov edi, 0x6fffffff
2. mov r10d, 0x6ffffdff
3. jmp 0x7f63eba6192a
4. cmp rax, 0x22
5. jbe 0x7f63eba61919
Trace 3: 5 Instructions, 2 CFI, Terminated due to max control flow instructions

5.3 Configurations

For this first analysis run using the following trace segmentation policy configurations:

Common for every run:

1. Max number of instructions: 32
2. Max number of control flow instructions: 4
3. Traces always terminate when encountering a Syscall.
4. Direct Unconditional Jumps do not count as a control flow instruction and do not terminate a trace.

Unique:
Every combination of configuration possible with the following control flow instructions terminating and not terminating a trace

1. Return
2. Direct Call
3. Indirect Call
4. Direct Conditional Jump
5. Indirect Unconditional Jump

A total of 2^5=32 different configurations for each benchmark.

The second experiment was done by taking the best configuration for each benchmark and increasing the number of instructions from 32 to 255.

The third experiment was done by again taking the best configuration for each benchmark, but direct conditional jumps are allowed. They count as 2 control flow instructions, which means that only 2 of them are allowed.

Finally, a fourth experiment, by splitting direct conditional jumps into 4 categories. Forward Taken, Forward Not Taken, Backward Taken and Backward Not Taken. After that run all 16 possible configurations with these 4 categories. Regarding the other control flow instructions, the configuration stayed the same as the best configuration from the first experiment.

5.4 Collected Data

For each configuration there exists every unique trace that occurred during execution along with the number of instructions and control flow instructions in the trace, the reason the trace terminated and the number of times the trace occurred.

5.5 Metrics

For comparison 3 metrics will be used:

1. Weighted Trace Length
2. Number of unique traces required to cover 90% of total instructions
3. Ratio of Max-Normalized Weighted Trace Length to Max-Normalized Number of Traces required to cover 90% of total instructions

For Average Trace Length Weighted:

$$Average\ Trace\ Length\ Weighted = \frac{\sum(Instructions_i \times Occurences_i)}{Occurences_i}$$

For Number of unique traces required to cover 90% of total instructions:

$$Total\ Coverage = \sum(Instructions_i \times Occurences_i)$$

$$Find\ minimum\ k\ such\ that$$

$$\sum_{i=1}^{k}(Instructions_i \times Occurences_i) \geq 0.9 \times Total\ Coverage$$

$$Trace\ Count = k$$

For Ratio of Max-Normalized Weighted Trace Length to Max-Normalized Number of Traces required to cover 90% of total instructions:

$$Let\ L_I = Length\ of\ trace\ i$$

$$Let\ C_I = Count\ of\ trace\ i$$

$$Let\ L_{max} = \text{Max}\ trace\ length\ configuration$$

$$Let\ C_{max} = Max\ trace\ count\ configuration$$

$$Max - Norm\ Ratio_i = \frac{\frac{L_i}{L_{max}}}{\frac{C_i}{C_{max}}}$$

For clarity these 3 metrics will be referred to as trace length, trace count and max-norm ratio.

The reason these metrics were chosen is because they respectively represent every aspect of a desirable trace set. The two things that are not considered here are data dependency as well as the number of unique traces. Since a trace cache is out of scope for the thesis the number of unique traces although it is an important metric will not be considered in this evaluation since all traces are stored.

The metric (Max-Normalized Weighted Trace Length) / (Max-Normalized Number of Traces required to cover 90% of total instructions) reflects every important quality of a trace set: Trace Length, Execution Frequency and Branch Consistency.

6 Chapter 6: Implementation

This chapter will explore how data was collected and stored during execution

6.1 Intel's Pin Tool

By using PIN's instrumentation which is performed by a just-in-time (JIT) compiler it is possible to analyze every instruction executed by running the binary files of each benchmark mentioned previously and monitor exactly every instruction executed one by one.
By instrumenting an instruction, a sort of "trap" is created and when the instruction gets execution it gets activated. Afterwards every time an instrumented instruction is encounter a specific function is called. Data regarding the instruction can be static and gathered during instrumentation or dynamic and gathered during runtime [6]. For example, what type of control flow instruction an instruction is investigated during instrumentation before execution but the outcome of a direct conditional jump (taken or not taken) is collected during execution.

6.2 Pin Tool for Trace Segmentation

The way traces are generated is by inspecting each instruction executed. When processing an instruction, it get added to the current whose state is stored separately, and then when it is terminated it is saved in a hash map that stores every trace executed so far. Once an instruction gets executed, the counter represents the number of instructions stored in the current trace increments. Afterwards a check happens if that instruction is a control flow instruction. If it is a control flow instruction, a counter representing the number of control flow instruction is incremented and then a check happens to determine what type of control flow instruction it is. The different types of control flow instruction under investigation are:

1. Syscall
   INS_IsSyscall(ins)

2. Return
   INS_IsRet(ins)

3. Direct/Indirect Call
   INS_IsCall(ins)) True if it is a Call
   INS_IsDirectControlFlow(ins) True if it is Direct

4. Direct/Indirect Conditional/Unconditional Jump forward/backward
   INS_IsBranch(ins) True if it is a Jump
   INS_HasFallThrough(ins) True if it is Direct
   INS_Address(ins) and INS_DirectControlFlowTargetAddress(ins) compare values to see if it is forward or backwards.

Now that type of instruction is determined a check is performed, to decide whether a trace should be terminated and stored or it should include more instructions. The conditions are as follows:

1. Exceeds Maximum number of instructions
2. Exceeds Maximum number of control flow instructions
3. The last control flow instruction is not allowed and terminates the trace

It is important to note that the control flow instruction that is responsible for terminating a trace is included in the trace as the last instruction.

If a trace does not terminate then it continues with the next instruction. If a trace does get terminated then it compares it with all the traces in the hash map to see if the same trace exists, if it does exist then it increments the counter representing the number of times that that trace has occurred, otherwise it adds to the hash map with a counter of 1.

6.2.1 Parameters

The pin tool takes input the following parameters

1. **-ins**: Maximum number of instructions that a trace is allowed to contain
2. -**cfis**: Maximum number of control flow instructions a trace is allowed to contain
3. -**flags**: an 11-bit mask, True if the control flow instruction is allowed to exist inside a trace, false if it can only be the last instruction of a trace.
4. -**start**: The number of instructions to start creating traces
5. -**stop**: The number of instructions to stop creating traces

This is an example input:

-ins 64 -cfis 8 -flags 1,0,1,1,0,1,1,1,0,0,1 -start 1000 -stop 2000

6.3 Trace Metadata

Since it is not possible to store every single instruction in a trace metadata is used instead.

1. Starting Address of the first instruction
2. Number of instructions in the trace
3. Number of control flow instruction in the trace
4. The reason the trace was terminated
5. An Incremental hash which is updated with the starting address of each basic block encountered in the trace

To avoid aliasing between structurally similar traces that follow different control paths, each basic block added to a trace (after the first) contributes its starting address to a incremental hash. This value acts as a lightweight signature of the trace's control path, ensuring trace uniqueness in the hash map. It is important to note that because the incremental trace hash is being used and not the complete addresses, it does not guarantee uniqueness. Different traces may occasionally produce the same hash value. There is a possibility that two distinct traces that share the same starting address, number of instructions, number of control flow instructions and reason terminated can produce the same hash value and be mistaken for the same the trace. It is unlikely but it is a compromise made to ensure more efficient memory usage.

6.4 Optimizations

With the goal of gathering as much data as possible and trying as many different configurations as possible it is important that Pin’s overhead was minimized, and execution times shortened. Having to analyze trillions of instructions was very difficult and time consuming so optimization regarding performance had to be made. Every optimization was tested using multiple types of different inputs to make sure to keep the originals integrity.

6.4.1 Trace Instrumentation

Pin's Trace Instrumentation has been a very powerful tool in terms of saving time during executions. PIN's Traces are made from basic blocks which can be instrumented instead, the term used for basic blocks inside of traces is BBL. Pin creates traces before execution by analyzing the instructions. Pin's traces are single entry, multiple exits meaning that a trace is guaranteed to always start from the first BBL but might exit beforehand. This means that it is not possible to instrument by trace since it is not guaranteed to be executed to its completion, but it is possible to instead instrument by BBLs. By instrumenting using BBLs it means that even if a trace (Pin's trace) exits early the BBLs that are included in the trace but did not get executed will not be accounted for.

6.4.2 Storing BBLs

During a block's instrumentation an object is created that contains the necessary info regarding the BBL. The info stored per BBL is saved in this object. It is important to note that this happens only during instrumentation, before execution. If the BBL executes during runtime the important info regarding the BBL is available which means that this object is created once for each block and then used every time that BBL gets executed.

```
struct BBLInfo {
   ADDRINT startAddress;
   std::vector<uint16_t> instructionOffsets;
   UINT8 reasonCode;
};
```

Storing addresses:

The starting address of the basic block is stored as well as the offsets from each instruction to the starting address. It is important that there is access to every address since when enforcing the trace segmentation policy there is a chance a trace terminates due to reaching the maximum number of instructions which would mean that a trace will be terminated mid BBL so there must always exist access to all the instructions' addresses. To save space due to the amount of BBLs an offset vector of type uint16 that records the offset of each address in the BBL with the starting address so that if need be, the address can be recreated without having to store

every single address of every single instruction. An attempt was made to use an 8-bit unsigned integer but the differences between addresses was longer which meant it was not possible.

Storing the last instruction:

During instrumentation the first time each BBL occurs what kind of instruction the last instruction is stored, its “tail”. The type of instruction the tail is, is stored using a code in an UINT8 so that as much space as possible is preserved as strings take up a lot of unnecessary space. Also important to note is that a BBLs tail is the only possible position in which a control flow instruction can exist, which means it is the only instruction that requires investigation.

Using only this data there is all the information required to process the BBL. The number of instructions that exist in a block can be determined by using the size of vector that stores the offset, every address in the BBL can be recreated, if need be, and information regarding what the last instruction is, are all stored and available at runtime. Also, this calculation only happens once and then every time the BBLs reappears the pointer containing that BBLs information is passed as an argument.

6.4.3 Storing Traces

To store traces, it is important to make sure they are stored using as little space as possible. To make the most out of memory, the smallest possible space can attribute are multiples of 8 bytes. The starting address of each trace itself is 8 bytes and type ADDRINT, which is a type of alias used in Intel Pin that represents an address-sized unsigned integer. So, trace metadata, aside from the address, are stored using only 7 bytes.

Using struct alignas(8) Trace and static_assert(sizeof(Trace) == 16 to assert that no memory space gets wasted and traces are only 16 bytes.

There exist 8 bytes available to allocated to necessary data

```
uint32_t branchOutcomes;     // 4 bytes
uint8_t instructionCount;    // 1 byte
uint8_t cfiCount;            // 1 byte
uint8_t reasonCode;          // 1 byte
uint8_t _padding = 0;        // to fill to 16 bytes
```

Space allocated is only 1 byte to instructions count, cfi count and reason code, which is the code representing the reason a trace terminated, since none of them will exceed 255 meaning a lot of space is saved by assigning them with uint8. 4 bytes are allocated to the value branchOutcomes, which is the incremental hash. Every starting address of a new BBL contributes to the hash to represent as a signature of its internal control flow.

To save the traces later in a hash map there exists custom hashing as well as equal operator function to decrease overhead as much as possible during lookups.

Every function is inline and noexcept to make them even faster since they are very simple and small functions. Although inline is not enforced and is still suggestive, even without inline, it still helps.

6.4.4 Trace HashMap

To data structure used to store traces is very important as the final result includes hundreds of thousands of unique traces. For performance-critical trace storage and lookup, a robin_hood::unordered_map was used [7], a high-performance hash map based on Robin Hood hashing and open addressing. It offers faster lookups and lower memory overhead compared to std::unordered_map, making it suitable for storing millions of trace entries.

Allocation for current trace happens only once so that there does not exist unnecessary memory allocation and deallocation

```
Trace globalTrace = {};
Trace* currentTrace = &globalTrace;
```

This is why there exists only 1 object current trace and it is access by a global pointer. Whenever a new trace needs to be created the current trace simply gets reset back to the original values.

When storing the trace in the hash map

```
auto [it, inserted] = traceCache.try_emplace(*currentTrace, 1);
    if (!inserted) it->second++;
```

Double lookups are avoided by first trying to place the current trace in the hash map. If the trace gets placed in hash map then it is copied by value and not address which means the current

trace pointer is reusable for the new current trace. If it already exists using the pointer by try_emplace the “second” which is the value representing the number of times this trace has occurred, gets incremented instead. Only 1 lookup required. Lastly, a large number of spaces are reserved beforehand in hash map to avoid needless rehashing.

7 Chapter 7: Results and Evaluation

This chapter will showcase the results and evaluation done of the experiments

7.1 500.perlbench

For the results these abbreviations will be used to represent the control flow instructions:
RET Return
DC Direct Call
IC Indirect Call
DCJ Direct Conditional Jump
IUJ Indirect Unconditional Jump

A first run was done which terminates a trace at every control flow instruction. This was done to gather information regarding each benchmarks type of control flow instruction and how often they appear.

code	DC (%)	DCJ (%)	DUJ (%)	IC (%)	IUJ (%)	RET (%)
500	5	75	9	1	5	6
502	8	75	7	1	1	8
503	1	89	6	0	1	1
505	0	73	7	9	0	10
507	4	27	4	0	1	4
508	0	36	3	0	0	0
510	0	97	2	0	0	0
511	9	59	12	5	0	14
519	0	19	6	0	0	0
520	8	64	4	6	3	14
521	6	75	5	0	0	6
523	0	93	3	2	0	2
525	2	57	6	2	0	3
526	2	83	11	0	0	2

527	4	79	6	0	2	4
531	13	65	8	0	0	13
538	0	99	0	0	0	0
541	14	65	4	0	0	14
544	5	77	7	0	2	5
548	0	87	12	0	0	0
549	0	0	0	0	0	0
554	3	78	4	0	2	3
557	1	81	13	0	2	1
Avg	4	67	6	1	1	5

Table 7.1: CFIs in Benchmarks

The first analysis will be regarding different configurations of what control flow instruction terminate the trace. Reminder that the configurations are all 32 combinations possible with the 5 control flow instructions.

Figure 7.1 shows the results of all 32 configurations of the 500.perlbench.
The x axis represents the number of unique traces ranked based on their coverage. The way a trace's coverage is calculated is by multiplying its length with the number of occurrences to show how many of the total instructions executed are covered by this trace. The trace with the highest coverage is the first one starting from 0. The x axis is also scaled logarithmically.
The y axis is the combined cumulative percentage coverage.

For example, in the point (x,y), y represents the percentage of the total instructions that are covered by the top x traces.

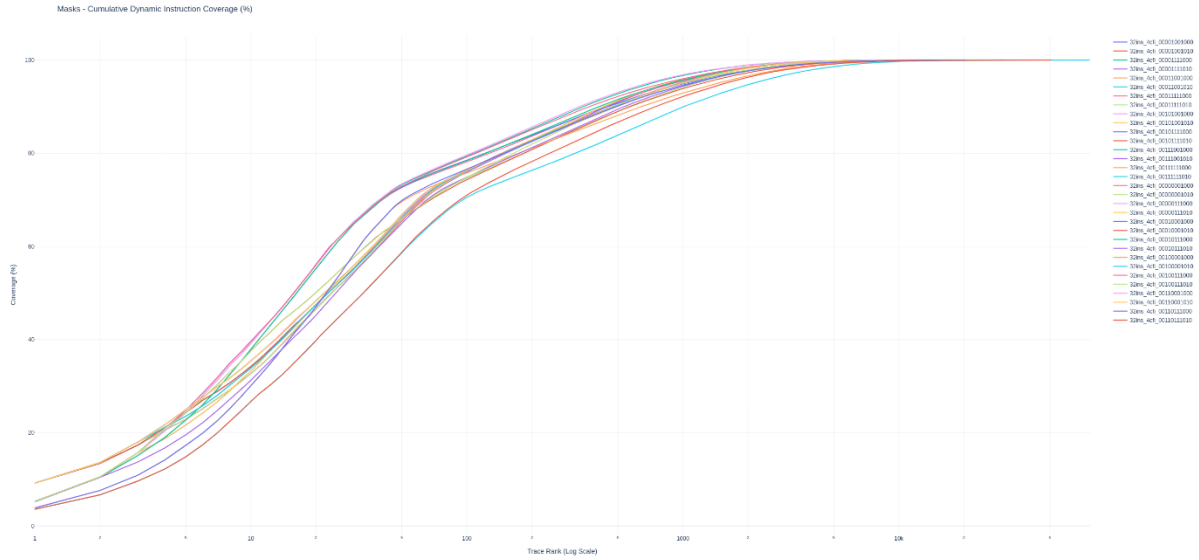

Figure 7.1: 500.perlbench results

Figure 7.1 shows a lot of overlaps and similar trends in the configuration, but it can be observed that between the percentages of 60% to 90% there is a configuration that covers more instructions with less unique traces. That being

RET 0

DC 0

IC 0

DCJ 1

IUJ 0

In this context 0 means not allowed to be included in a trace and terminates it and 1 means it is allowed in the trace, and the trace can go on and include the next instruction as well.

To better quantify this the 2 metrics are used instead. These being weighted trace length (trace length) and the number of unique traces required to cover 90% of total instructions executed (trace count)

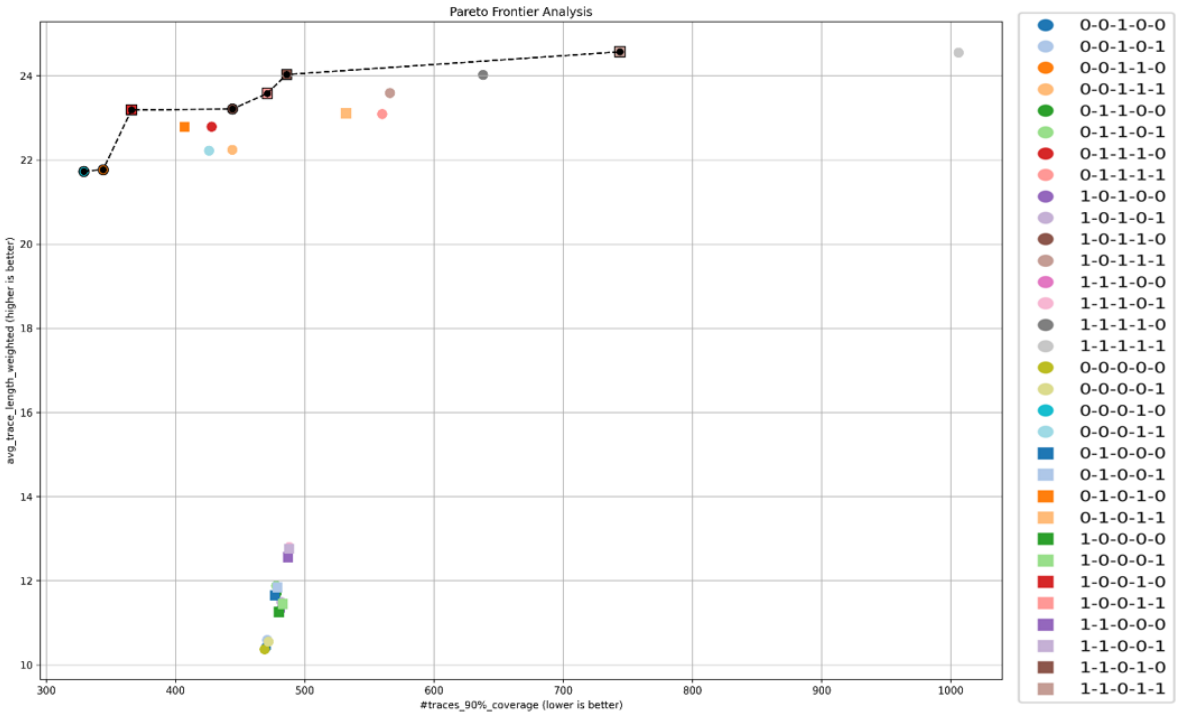

Figure 7.2: 500.perlbench Trace Length + Count

Figure 7.2 plots the results of the 500.perlbench with the x axis being trace count and y axis being trace length. In this example a configuration that is in the top left corner would be the best one since it would have the longest trace length and the smallest trace count which is after more desirable outcome. As indicated by Figure 7.2 there is an important trade-off amongst the configurations. There is not one single configuration that has the longest trace length and the smallest trace count. The configurations marked with the dotted line are pareto rank 0 configurations essentially meaning that there does not exist a configuration that has both a bigger trace length and a smaller trace count. Based on the system and demands all 7 of those configurations could be considered an optimal choice. There needs to be a decision to decide which of the two metrics takes priority and to what degree. In any case, these 7 configurations are the best results for 500.perlbench and every other result should not be considered.

To make this trade off quantifying the metric max-norm ratio will be used in order to combine the 2 metrics into one quantifiable metric to use as reference for comparison.

Figure 7.3 shows the relationship between Trace Count and Trace Length in regard to how these values affect the max-norm ratio. Circled in blue are the values representing Trace Length and circled in red are the values representing Trace Count. The plots are ranked from lowest max-norm ratio to highest. This Figure is meant to indicate how these two values contribute when combined to make the max-norm ratio. To give a visual representation of what is considered a better combination of these 2 metrics.

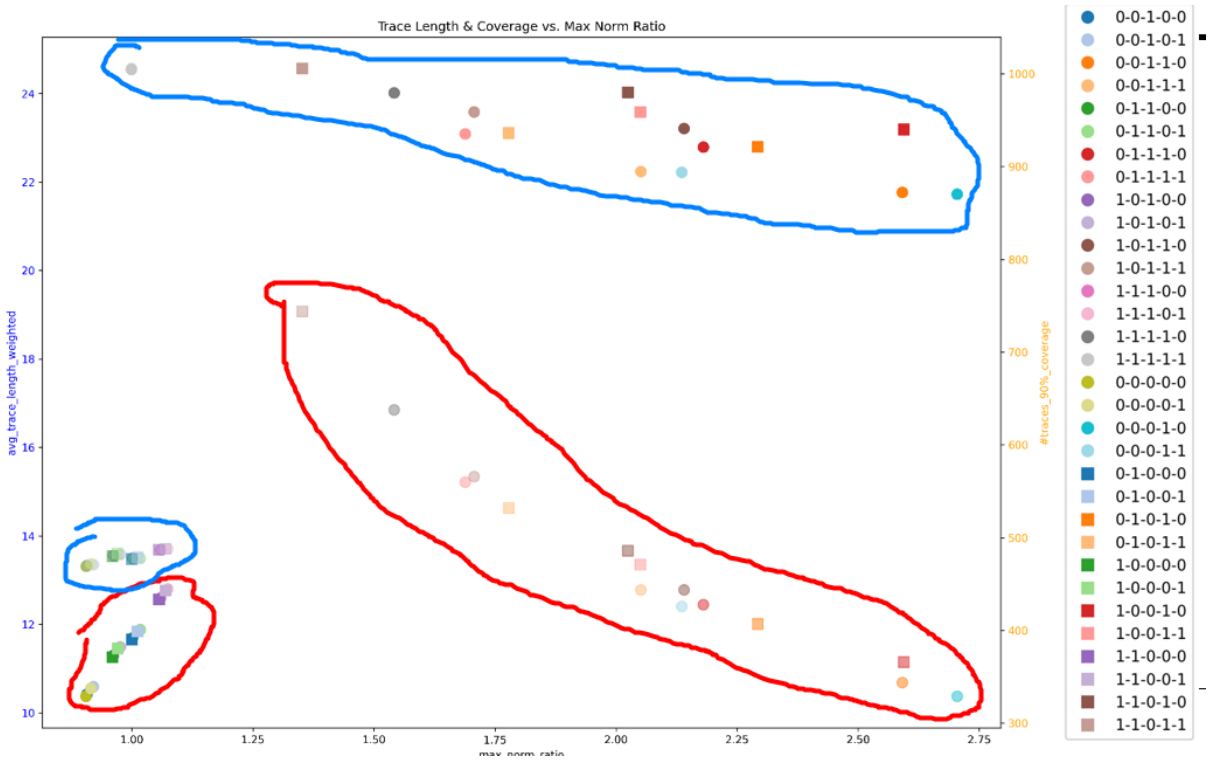

Figure 7.3: 500.perlbench Max-Norm Ratio

7.2 Control Flow Instructions Analysis

Table 7.2 shows the best configuration for each benchmark according to its max-norm ratio.

filename	RET	DC	IC	DCJ	IUJ	Trace Length	Trace Count	Max-Norm Ratio
500.perlbench_r	0	0	0	1	0	21.73	329	2.7
502.gcc_r	0	0	0	1	0	17.99	9984	3.11
503.bwaves_r	0	0	1	1	1	27.62	167	1.53
505.mcf_r	0	0	0	1	0	16.6	52	3.5
507.cactuBSSN_r	1	1	1	0	1	30.12	403	6.91
508.namd_r	1	1	1	0	1	29.61	106	25.09
510.parest_r	1	0	1	1	1	29.97	74	1.27
511.povray_r	0	0	1	1	1	22.35	210	1.88
519.lbm_r	0	0	1	0	0	31.16	13	31.98
520.omnetpp_r	1	0	1	1	0	19.73	227	2.07
521.wrf_r	1	1	0	0	0	21.45	1295	4.44
523.xalancbmk_r	0	0	0	1	1	13.51	23	2.08
525.x264_r	0	0	0	0	0	25.2	294	7.7
526.blender_r	1	1	1	0	0	14.24	62	1.93
527.cam4_r	1	0	1	0	1	15.89	1600	2.1
531.deepsjeng_r	0	0	1	1	1	21.73	520	3.12
538.imagick_r	0	1	1	0	0	18.58	1	1.76
541.leela_r	1	1	1	0	1	16.14	249	2.27
544.nab_r	0	1	1	1	0	28.82	51	1.37
548.exchange2_r	1	1	1	0	0	16.67	293	2.26
549.fotonik3d_r	1	1	1	0	0	28.77	103	19.41
554.roms_r	1	1	1	0	1	20.39	313	7.17
557.xz_r	1	1	1	0	1	15.91	132	2.52

Table 7.2: Best Configurations

Table 7.3 shows the number of times each type of control flow instruction appeared to be allowed and allowed

Type	Include (1)	Terminate (0)
Return	12	11
Direct Call	11	12
Indirect Call	17	6
Direct Conditional Jump	10	13
Indirect Unconditional Jump	11	12

Table 7.3: CFIs terminate or not

There does not appear to be a distinct instruction that should always terminate or not terminate a trace. The most common result has been by a bit allowing indirect calls which appeared in 17 out of 23 benchmarks, for the other instructions there does not appear such a strong case although this data does suggest that direct conditional jumps should most commonly terminate a trace.

Table 7.4: Show the 13 most common configurations that yielded the best results.

Return	Direct Call	Indirect Call	Direct Conditional Jump	Indirect Unconditional Jump	Count
1	1	1	0	1	5
0	0	1	1	0	3
0	1	1	1	0	3
1	1	1	0	0	3
1	0	1	1	1	1
0	1	0	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1
0	0	0	1	1	1
1	0	0	1	0	1
0	1	1	0	1	1
0	1	1	1	0	1

Table 7.4: Common Patterns

The most common configuration is shown in Table 7.5

Return	Direct Call	Indirect Call	Direct Conditional Jump	Indirect Unconditional Jump	Count
1	1	1	0	1	5

Table 7.5: Most common config

This configuration is the best configuration for most benchmarks. Another indication is that Direct Conditional Jumps should terminate a trace. Also, interestingly enough the top 5 most common configurations all include Indirect Calls.

Table 7.6 shows the average Hamming distance of every possible combination of configurations to the best configuration ranked from best to worst.

Return	Direct Call	Indirect Call	Direct Conditional Jump	Indirect Unconditional Jump	Average Hamming Distance
1	0	1	0	0	2.13
1	1	1	0	0	2.17
0	0	1	0	0	2.17
1	0	1	0	1	2.17
1	1	1	0	1	2.22
0	0	1	0	1	2.22
0	1	1	0	0	2.22
1	0	1	1	0	2.26
0	1	1	0	1	2.26
1	0	1	1	1	2.3
0	0	1	1	0	2.3
1	1	1	1	0	2.3
1	1	1	1	1	2.35
0	0	1	1	1	2.35
0	1	1	1	0	2.35
0	1	1	1	1	2.39
1	0	0	0	0	2.61

1	0	0	0	1	2.65
1	1	0	0	0	2.65
0	0	0	0	0	2.65
0	1	0	0	0	2.7
1	1	0	0	1	2.7
0	0	0	0	1	2.7
1	0	0	1	0	2.74
0	1	0	0	1	2.74
1	0	0	1	1	2.78
1	1	0	1	0	2.78
0	0	0	1	0	2.78
0	1	0	1	0	2.83
0	0	0	1	1	2.83
1	1	0	1	1	2.83
0	1	0	1	1	2.87

Table 7.6: Hamming Distance

Again, the data suggests that Conditional Jumps should terminate a trace and Indirect Calls should not. The top 7 results all have conditional jumps not allowed and indirect calls allowed.

The configuration shown in Table 7.7 is the most neutral configuration that is as close as to the best configuration out of every single benchmark as possible.

Return	Direct Call	Indirect Call	Direct Conditional Jump	Indirect Unconditional Jump	Average Hamming Distance
1	0	1	0	0	2.13

Table 7.7: Best Hamming Distance

Now these comparisons regarding max-norm ratio were done in isolation within the benchmark. It is important to note though that max-norm ratio when calculating in the context of a single benchmark not only represents what configuration is best, but it also quantifies just how much better than configuration is. For example, Table 7.8 shows the best configurations

for 510.parest_r and 519.lbl_r, but the values of the max-norm ratio are vastly different. This is the case because for 510.parest_r every configuration was very close, but this one proved to be the best. For 519.lbm_r on the other hand different configurations resulted in vastly different outputs and this configuration was a significant improvement among the others.

This is an important observation to make when comparing what the best configuration is overall using max-norm ratio, since now the impact it had overall compared to other configurations will come into play.

filename	RET	DC	IC	DCJ	IUJ	Trace Length	Trace Count	Max-Norm Ratio
510.parest_r	1	0	1	1	1	29.97	74	1.27
519.lbm_r	0	0	1	0	0	31.16	13	31.98

Table 7.8: Lowest, Highest Max-Norm Ratio

Figure 7.4 shows exactly the concept mentioned previously. The benchmarks are ranked from left to write based Max-Norm Ratio of the best configuration of that benchmark. What this means is that this particular configuration played a small role in benchmarks that appear on the left but make a significant impact on the benchmarks that appear on the right.

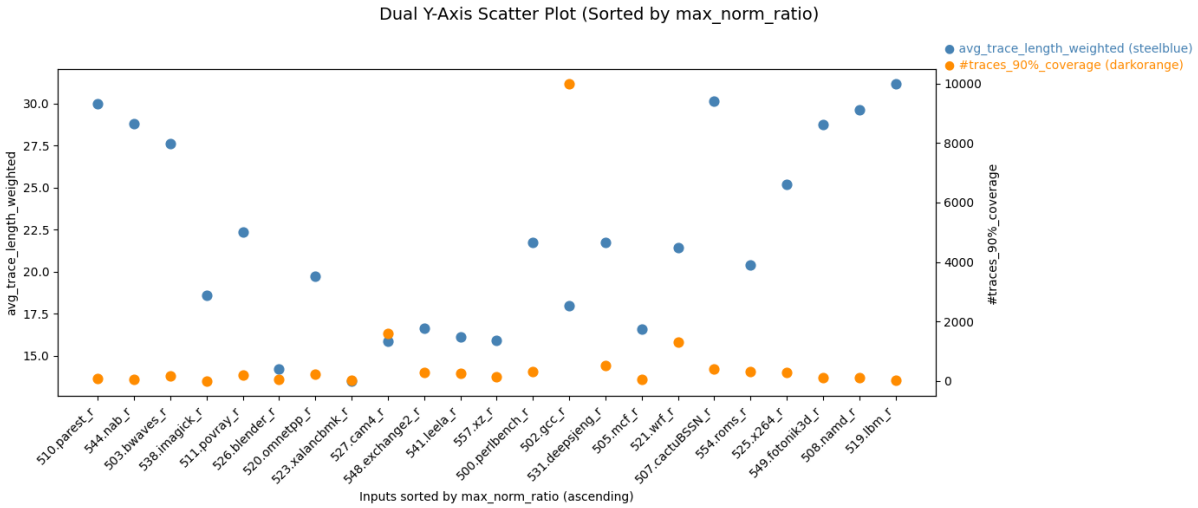

Figure 7.4: Winner ranked by Max-Norm Ratio

It can be observed from Figure 7.4 that there is essentially 3 types of benchmarks. The number of instructions following a sort of upside-down bell distribution and so categorizes the benchmarks into 3 different types.

Benchmarks on the left have infrequent and predictable branches. This is shown since max norm ration is very small, suggesting that different configurations did not have impact on the results all that much and the traces are very long, almost at the limit of 32. Which means that traces very frequently terminated simply because they reached the limit.
Benchmarks in the middle suggest very frequent and unpredictable branching. Traces are the shortest here and it can even observed that an increase in the number of traces required for 90% coverage. Different configurations did have an impact here, but the unpredictability of the branches resulted in a minor increase in quality of traces.

Lastly, the benchmarks on the left are the ones that had the most impact. Here different configurations made an impact and the traces are longer. This suggests that the benchmarks here have frequent yet more predictable branching. It shows how including predictable branches and excluding unpredictable ones have increased the quality of the trace set by a significant margin.

Table 7.9 shows the average Max-Norm Ratio score of all 32 configurations tested across al benchmarks

RET	DC	IC	DCJ	IUJ	Max-Norm Ratio
1	1	1	0	1	5.46
1	1	1	0	0	5.45
1	0	1	0	1	5.44
1	0	1	0	0	5.43
0	1	1	0	1	5.43
0	1	1	0	0	5.41
0	0	1	0	1	5.39
0	0	1	0	0	5.38
1	1	0	0	1	4.92

1	1	0	0	0	4.90
0	1	0	0	1	4.85
1	0	0	0	1	4.85
1	0	0	0	0	4.83
0	1	0	0	0	4.82
0	0	0	0	1	4.78
0	0	0	0	0	4.76
0	0	0	1	0	2.24
0	0	0	1	1	2.23
1	0	0	1	0	2.10
1	0	0	1	1	2.08
0	1	0	1	0	2.05
0	1	0	1	1	2.01
1	1	0	1	0	1.64
0	0	1	1	0	1.61
0	0	1	1	1	1.58
1	1	0	1	1	1.56
0	1	1	1	0	1.47
1	0	1	1	0	1.41
0	1	1	1	1	1.41
1	0	1	1	1	1.39
1	1	1	1	0	1.09
1	1	1	1	1	1.00

Table 7.9: Max-Norm Ratio Ranked

From these two tables it can be deduced that Direct Conditional Jumps have made the most impact suggesting that including them in traces can degrade the overall quality of a trace set. Every single configuration that includes direct conditional jumps is worse than every single configuration that does not.

Again a repeating pattern is shown regarding indirect calls suggesting that indirect calls should be included in traces, as the top 8 configuration all include indirect calls in the trace. Important to note is the fact that the overall impact appears to be very little. But it is quite polarizing since

Indirect Call is included in the best configuration but also in the worst, not the configurations in between.
The data also suggests that returns and direct calls should also be included and including them also has an equivalent impact on trace sets as whole, but there does not seem to be as clear of a pattern as direct conditional jumps.
Lastly Indirect Unconditional jumps have the least impact and even seem to follow a random pattern in the rankings, making it the least impactful and insignificant control flow instruction.

Using the best configuration for each benchmark individually from the previous analysis, as a base of what control flow instructions to terminate and not terminate a trace at another experiment took place this time altering the number of instructions from 32 to 255.

7.3 Instructions Analysis

Table 7.10 shows the difference between the best configuration for that benchmark with 32 instructions and then the same configuration but instead with a limit of 255 instructions this time. The difference was 32-255.

name	Δ Trace Length	Δ Trace Count	Max-Norm Ratio 32	Max-Norm Ratio 255	Δ Max-Norm Ratio
500	3.52	22	1.00	0.90	0.10
502	3.98	309	1.00	0.80	0.20
503	0.68	69	1.00	1.66	-0.66
505	3.34	6	1.00	0.90	0.10
507	-34.98	339	0.46	6.30	-5.83
508	-38.71	75	0.43	3.42	-2.99
510	-0.66	19	0.98	1.35	-0.37
511	3.71	68	1.00	1.23	-0.23
519	-77.89	10	0.29	4.33	-4.05
520	3.71	3	1.00	0.82	0.18
521	8.97	332	1.00	0.78	0.22
523	0.64	-1	1.04	0.95	0.09
525	1.11	183	1.00	2.53	-1.53

526	7.25	7	1.00	0.55	0.45
527	6.81	160	1.00	0.63	0.37
531	3.74	101	1.00	1.03	-0.03
538	6.88	-1	2.00	0.63	1.37
541	7.39	13	1.00	0.57	0.43
544	-6.71	20	0.81	1.65	-0.83
545	6.83	9	1.00	0.61	0.39
549	-23.61	77	0.55	3.96	-3.41
554	4.70	51	1.00	0.92	0.08
557	6.89	18	1.00	0.66	0.34
Averages	-4.45	82.13	0.94	1.62	-0.68

Table 7.10: 32 vs 255 Instructions

Trace Length:

- Improvement: 6 out 23 benchmarks
- Average Improvement per Benchmark: 4.45

Trace Count:

- Improvement: 21 out 23 benchmarks
- Average Improvement per Benchmark: 82.13

Incredible improvement in Trace Count when increasing the number of instructions. The 2 benchmarks that did not improve had a difference of 1 trace only. The hot traces are even better which comes to show that hot spots in the code are often segmented by control flow instructions and number of instructions should not limit these traces.

Average Max-Norm-Ratio:

- 32 Instructions: 0.94
- 255 Instructions: 1.62

Overall improvement, there seems to be no indication that a trace should segment based on number of instructions. This being the case even if there no mechanism to detect where other traces begin when segmenting. Even if there is overlap between traces that share the same instruction the overall improvement is undeniable.

7.4 Direct Conditional Jumps Analysis

In this analysis two types of configurations will be tested. The first one will be comparing the best results but allow direct conditional jumps. This time though Direct Conditional Jumps will count as 2 control flow instruction essentially meaning that only 2 are allowed per trace.
The second one will be breaking down direct conditional jumps into 4 categories. Those categories being forward/backwards as well as taken and not taken. This aims to find if some types of direct conditional jumps are more consistent than others with the hopes that traces will improve if only certain types of direct conditional jumps terminate them.

7.4.1 Two Direct Conditional Jumps

A third analysis took place again using the best configuration for each benchmark individually as a base. This time everything is the same as the first analysis, but traces are only limited to 2 conditional branches, and conditional branches are always allowed. The theory being that if 4 conditional branches are too inconsistent to include in a trace maybe including only 2 might show some improvement.
To clarify the comparison was made by simply leaving everything just like the best configuration same instructions, control flow instructions and terminated control flow instructions but the only change is direct conditional jumps. In this run direct conditional jumps are allowed for all configurations but only 2 of them. This is regardless of if the best configuration included DCJ or not. See Table 7.11

name	Δ Trace Length	Δ Trace Count	Max-Norm Ratio 4 DCJ	Max-Norm Ratio 2 DCJ	Δ Max-Norm Ratio
500	11.14	-44	1.13	0.49	0.65
502	9.61	-368	1.04	0.47	0.57
503	13.74	90	1.00	1.09	-0.09
505	9.11	6	1.00	0.51	0.49

507	1.39	-2133	6.29	0.95	5.34
508	-1.14	-1405	13.73	1.00	12.73
510	14.56	38	1.00	1.06	-0.06
511	10.01	25	1.00	0.63	0.37
519	0.57	-1	1.08	0.98	0.10
520	9.17	-27	1.12	0.54	0.58
521	2.72	-1637	2.26	0.87	1.39
523	6.86	-4	1.17	0.49	0.68
525	0.74	-395	2.34	0.97	1.37
526	1.00	-27	1.44	0.93	0.51
527	0.31	-475	1.30	0.98	0.32
531	9.80	94	1.00	0.67	0.33
538	-4.74	-1	1.59	1.00	0.59
541	2.01	-140	1.56	0.88	0.69
544	11.59	2	1.00	0.62	0.38
548	-0.99	-160	1.46	1.00	0.46
549	-1.96	-319	3.84	1.00	2.84
554	-2.72	-681	2.80	1.00	1.80
557	-0.61	-122	1.85	1.00	0.85
Averages	4.44	-334.09	2.26	0.83	1.43

Table 7.11: 4 CJ vs 2 CJ

Trace Length:

- Improvement: 6 out 23 benchmarks
- Average Improvement per Benchmark: -4.44

Overall Trace Length is shorter when including 2 DCJ

Trace Count:

- Improvement: 6 out 23 benchmarks
- Average Improvement per Benchmark: -344.09

Overall Trace Count is also much smaller when including 2 DCJ

Average Max-Norm-Ratio:

- UP TO 4 DCJ: 2.26
- ONLY 2 DCJ: 1.43

It appears that the problem with the initial configurations that did not include direct conditional jumps is not the fact that 4 control flow instructions got too inconsistent. It appears that even 2 direct conditional jumps are not worth including a trace. Trace should always terminate when encounter a direct conditional jump if there is no prior history of that conditional jump being consistent.

7.4.2 Types of Direct Conditional Jumps

For this analysis Direct Conditional Jumps have been classified by their direction. The two distinctions made for their direction is whether they were taken and not taken, meaning their condition was true or false, and the relative position of their target in relation to their own position, meaning the jump was backwards or forwards in terms of addresses.

For clarity the 4 categories will be:

- Forward Taken
- Forward Not Taken
- Backward Taken
- Backward Not Taken

The configuration for instructions and control flow instructions is the same as the first analysis as to keep consistency in the experiment. As per the first analysis the number of instructions allowed was 32 and control flow instructions was 4. For the other 4 control flow instructions, which are Return, Direct Call, Indirect Call and Indirect Unconditional Jump, they remained the same based on the best configuration made in the first analysis. Now using their best configuration from the best analysis each benchmark was tested using all 16 combinations of the 4 direct conditional jumps.

Table 7.12 Shows the best configuration out of 16 for each benchmark individually

Code	Forward Taken	Forward Not Taken	Backward Taken	Backward Not Taken	Trace Length	Trace Count	Max Norm Ratio
500	0	1	1	1	17.06	328	1.43
502	1	1	1	1	13.76	9982	1.19
503	1	1	1	0	10.26	55	1.3
505	1	1	1	1	13	53	1.06
507	1	0	1	1	26.58	423	5.94
508	1	1	1	0	27.9	118	21.28
510	1	0	1	1	23.37	54	1.14
511	0	1	0	1	16.54	211	1
519	0	1	0	0	30.6	14	29.24
520	0	1	0	1	14.83	215	1.4
521	1	1	1	0	20.66	1364	4.4
523	0	1	0	1	12.87	24	1.67
525	0	1	0	0	20.59	278	2.11
526	1	1	0	0	17.24	114	1.48
527	0	1	1	0	15.99	1364	2.21
531	1	1	1	0	13.01	393	1.05
538	0	1	0	0	23.59	1	3
541	0	1	1	0	15.32	281	2.13
544	1	1	1	0	23.08	37	1.32
548	1	1	0	0	19.51	253	3.26
549	0	0	1	0	25.84	104	16.93
554	0	1	1	0	18.44	336	6.25
557	0	1	1	0	18.67	158	2.63

Table 7.12:Best DCJ F/B,T/NT

An improvement can most certainly be seen since none of the benchmarks' best configuration was 0,0,0,0. This proves to show that although the majority of the benchmarks' best configuration in the first analysis had Direct Conditional Jumps terminate the trace, even more importantly the analysis shown on 7.4.1 which included only 2 direct conditional jumps showed that it did not improve results. Based on those 2 findings one might assume that direct conditional jumps should always terminate a trace, but Table 7.11 conflicts with the idea as when breaking down direct conditional jumps into these 4 categories none of the benchmarks' best configuration was to terminate at every direct conditional jump. Every single benchmark run shows that it benefits from at least 1 type of DCJ being included.

Table 7.13 shows all 7 unique combinations out of the best configurations for each benchmark, along with the number of times each combination appeared.

Forward Taken	Forward Not Taken	Backward Taken	Backward Not Taken	Count
1	1	1	0	5
0	1	1	0	4
0	1	0	1	3
0	1	0	0	3
1	1	1	1	2
1	0	1	1	2
1	1	0	0	2
0	1	1	1	1
0	0	1	0	1

Table 7.13: Common Combinations F/B T/NT

Most common 9 combinations included Backward Taken and terminated at Backward Not Taken, showing that backwards taken are much more consistently. This could possibly be explained since backward branches are typically used in loops which makes them valuable to include in traces when they are taken. Also, the most common 17 combinations show that including Forward Not Taken branches has benefited them.

For more comparison Table 7.14 shows the number of times each type of Direct Conditional Jump was included or terminated in the best combinations.

Type	Included (1)	Terminated (0)
Forward Taken	11	12
Forward Not Taken	20	3
Backward Taken	15	8
Backward Not Taken	8	15

Table 7.14: Included and Terminated F/B, T/NT

Table 7.14 shows even more clearly the comparison between Backward Taken and Not Taken. 15 out of 23 benchmarks shows better results when including Backward Taken and terminating

when Backward Not Taken. The data for Forward Taken is not as clear as the comparisons are very close but Forward Not Taken shows by far the biggest improvement when included.

Table 7.15 showcases all 16 configurations ranked by average Max-Norm Ratio

Forward Taken	Forward Not Taken	Backwards Taken	Backward Not Taken	Max-Norm Ratio
1	1	1	0	4.63
0	1	1	0	4.62
1	1	0	0	4.61
0	1	0	0	4.6
1	0	1	0	4
0	0	1	0	4
1	0	0	0	3.99
0	0	0	0	3.98
1	0	1	1	2.68
0	0	1	1	2.67
1	0	0	1	2.67
0	0	0	1	2.66
1	1	0	1	1.08
1	1	1	1	1.08
0	1	0	1	1.07
0	1	1	1	1.07

Table 7.15: Max-Norm Ratio F/B, T/NT

The most definitive conclusion that can be deduced is the fact that Backward Not Taken DCJ should always terminate trace. Every configuration that does not allow Backward Not Taken DCJ resulted in a better score than those who did. There also appears to be strong evidence again that allowing Forward Not Taken DCJ also yields better results. A common pattern appears as well from the previous analysis. The fact that more lenient configurations made for an overall better trace sets but the differences here are actually very close in comparison. The top 4 configurations have almost the same Max-Norm Ratio and the only consistency between all 4 is Forward Not Taken allowed and Backward Not Taken not allowed.

8 Chapter 8: Conclusion & Future Work

8.1 Conclusion

After running these experiments, it appears that the number one factor most important quality of trace must have is branch consistency. Branch consistency has proven to be incredibly impactful and creates much worse trace sets with a lot more overlap. If traces do not have consistent branches inside them the number of unique traces increase dramatically and the frequency of execution drops significantly.

Direct Conditional Jumps require more information regarding their consistency to be included in a trace. If no such information exists such as history and a pattern, then direct conditional jumps should always terminate a trace. Even 2 direct conditional jumps in the same trace appear to significantly degrade the quality of the trace set since the traces are no longer branch consistent. If runtime patterns are not available during trace generation though it appears that segmenting on Backwards Not Taken and including Forward Not Taken is a good guideline. Backwards Taken Branch also show a lot of consistency since a lot of the times they are implemented they are part of a loop which means that they are very consistent.

There appears to be multiple evidence that indirect calls should not terminate a trace but instead be allowed to be included in the trace. This is the most confident and consistently reappearing rule that yielded the best results. Configuration with Indirect Call allowed although it did not show drastic improvement over the others, they did increase the quality just a little bit yet a lot more consistently than other control flow instructions.

Having a Trace Set with better hot traces is much more valuable, even at the cost of worst cold traces. A few traces that cover almost all the total instructions is the easiest way to identify a good trace set.

Greater Trace Length achieved by more “lenient” configurations appear to benefit much more than “stricter” configurations with frequent termination. A less frequent termination configuration results in more overlap between traces but the trade-off of creating longer traces

improves trace quality by a significant margin so it appears that longer traces with overlap are better, but they need to have consistent branching.

Trace segmentation based on the number of instructions has not improved results, even if there is no mechanism to detect the start of other traces when deciding when to terminate one. The overlap between traces due to high instruction counts is worth it since instructions that do not introduce unpredictability are great to have in a trace.

8.2 Improvements and Limitations

This evaluation was done using speculative metrics. A more objective evaluation would be to implement a trace predictor and trace cache in a simulated environment where execution-based metric collected at run time would provide a clearer picture as to what configurations are objectively better

The workload run using the benchmarks might not be representative of common coding patterns, thus making the results too specific to these benchmarks.

There does exist aliasing between traces as their Trace ID is not completely unique and there exists a possibility where 2 distinct traces are mistaken as one.

Data dependencies as well as the number of unique traces were not taken into consideration during evaluation.

8.3 Future work

A trace segmentation policy such as the one explored in this thesis can be used when encountering new control flow instructions that there does not exist a definitive way to know how predictable that control flow instructions. If more information can be obtained during the run, then a method to restructure existing traces would be a very strong trace segmentation policy. A policy which dynamically changes during runtime based on previous patterns.

Another important aspect would be to find a mechanism to minimize unnecessary overlap but keep trace lengths as large as possible

A metric to detect branch consistency during execution could be used instead of a simple incremental counter every time a control flow instruction occurs. This metric would exist to quantify how consistently each basic block in a trace follows the direction of the trace. Then to calculate an overall trace consistency, one could calculate the product of all the branch consistencies between all the basic blocks. For example, blocks A, B and C exist in a trace and

block A goes to B 90% of them time and block B goes to C 20% the trace would have a branch consistency of 0.9 x 0.2 = 0.18. This would be a better representation of when to terminate a trace than a simple counter.

Since long and overlapping traces appear to improve the trace set a more sophisticated trace cache would be very beneficial which takes into consideration not only occurrences of a the whole trace itself but use a hybrid approach where occurrences of basic blocks inside the trace are also taken into consideration. This hybrid approach would allow for longer traces with overlap to be correctly stored without competing for a spot in the cache.

9 Bibliography

[1] A. Yasin, "A Top-Down method for performance analysis and counters architecture," 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459.

[2] E. Rotenberg, S. Bennett and J.E. Smith, “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching”, in MICRO29, Dec. 1996.

[3] Q. Jacobson, E. Rotenberg and J. E. Smith, "Path-based next trace prediction," *Proceedings of 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, USA, 1997

[4] E. Rotenberg, S. Bennett and J. E. Smith, "A trace cache microarchitecture and evaluation," in IEEE Transactions on Computers, vol. 48, no. 2, pp. 111-120, Feb. 1999

[5] Roni Rosner, Micha Moffie, Yiannakis Sazeides, and Ronny Ronen. 2003. Selecting long atomic traces for high coverage. In Proceedings of the 17th annual international conference on Supercomputing (ICS '03). Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/782814.782818

[6] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[7] M. Ankerl, robin-hood-hashing, GitHub repository, https://github.com/martinus/robin-hood-hashing, Accessed: May 25, 2025.

10 Appendix A

Trace Segmentation Pin Tool

```
#include "pin.H"
#include <vector>
#include <fstream>
#include <iostream>
#include <sstream>
#include "Trace.h"
#include "robin_hood.h"


// Trace occurrences cache
robin_hood::unordered_map<Trace, uint64_t> traceCache;

KNOB<UINT32> knobMaxInstructions(KNOB_MODE_WRITEONCE, "pintool", "ins", "32", "Maximum instructions per trace");
KNOB<UINT32> knobmaxCFIs(KNOB_MODE_WRITEONCE, "pintool", "cfis", "4", "Maximum control-flow instructions per trace");
KNOB<std::string> knobReasons(KNOB_MODE_WRITEONCE, "pintool", "flags", "1,1,1,1,1,1,1,1,1,1,1,1", "Comma-separated reason toggles");
KNOB<UINT64> knobStartInstr(KNOB_MODE_WRITEONCE, "pintool", "start", "0", "Start collecting traces after this instruction count");
KNOB<UINT64> knobStopInstr(KNOB_MODE_WRITEONCE, "pintool", "stop", "0", "Stop collecting traces after this instruction count (0 = no stop)");

UINT64 instructionCounter = 0;
UINT64 startInstr = 0;
UINT64 stopInstr = 0;

int maxInstructions;
int maxCFIs;
bool reasonEnabled[12] = {false};
std::string outputFileName;
```

```
Trace globalTrace = {};
Trace* currentTrace = &globalTrace;

struct BBLInfo {
    ADDRINT startAddress;
    std::vector<uint16_t> instructionOffsets;
    UINT8 reasonCode;
};

std::vector<BBLInfo*> allocatedBBLs;  // for memory cleanup

UINT8 GetReasonCode(INS ins) {

    if (INS_IsSyscall(ins)) return 1;       // Syscall
    if (INS_IsInterrupt(ins)) return 2;     // Interrupt
    if (!INS_IsControlFlow(ins)) return 0;  // Not a CFI
    if (INS_IsRet(ins)) return 3;           // Return

    if (INS_IsCall(ins)) {
        if (INS_IsDirectControlFlow(ins)) return 4;  // Direct Call
        else return 5;                               // Indirect Call
    }

    if (INS_IsBranch(ins)) {
        if (INS_IsDirectControlFlow(ins)) {
            if (INS_HasFallThrough(ins)) {
                // Conditional Direct Jump
                ADDRINT target = INS_DirectControlFlowTargetAddress(ins);
                ADDRINT ip = INS_Address(ins);
                if (target > ip) return 6;  // Conditional Direct Jump Forward
                else return 7;              // Conditional Direct Jump Backward
            } else {
                return 8;  // Unconditional Direct Jump
```

```
            }
        } else {
            if (INS_HasFallThrough(ins)) return 9;  // Conditional Indirect Jump
            else return 10;                         // Unconditional Indirect Jump
        }
    }

    return 11; // Unknown CFI
}



inline void CommitTrace() {
    if (currentTrace->getInstructionCount() > 0) {
        auto [it, inserted] = traceCache.try_emplace(*currentTrace, 1);
        if (!inserted) it->second++;
        currentTrace->reset();
    }
}

ADDRINT GetInstructionAddress(BBLInfo* bblInfo, int index) {
    return bblInfo->startAddress + bblInfo->instructionOffsets[index];
}

VOID LogBBL(BBLInfo* bblInfo, BOOL branchTaken) {

    instructionCounter+=bblInfo->instructionOffsets.size();
    if(startInstr != 0 && instructionCounter<startInstr){
        return;
    }else if(stopInstr != 0 && instructionCounter>stopInstr){
        PIN_ExitApplication(0);
    }

    int currentIns = bblInfo->instructionOffsets.size();
```

```
int index = 0;

if (currentTrace->getInstructionCount() == 0) {
   currentTrace->startAddress = bblInfo->startAddress;
} else {
   currentTrace->recordFallthroughAddress(bblInfo->startAddress);
}

while (currentIns + currentTrace->getInstructionCount() > maxInstructions) {
   int available = maxInstructions - currentTrace->getInstructionCount();
   currentTrace->instructionCount = maxInstructions;
   currentTrace->reasonCode = 12; // max instructions reached
   CommitTrace();

   currentIns -= available;
   index += available;

   currentTrace->startAddress = GetInstructionAddress(bblInfo, index);
}

currentTrace->instructionCount += currentIns;

UINT8 reasonCode=bblInfo->reasonCode;

if (reasonCode != 0) {

   if (reasonCode!= 8)
      currentTrace->cfiCount++;

   if(reasonCode == 6 && branchTaken){
      reasonCode=2;
   }
   else if(reasonCode == 7 && !branchTaken){
      reasonCode=11;
```

```
        }

        if (!reasonEnabled[reasonCode]) {
            currentTrace->reasonCode = reasonCode;
            CommitTrace();
        } else if (currentTrace->cfiCount >= maxCFIs) {
            currentTrace->reasonCode = 13; // max allowed control-flow instructions reached
            CommitTrace();
        }
    }
}

std::string ReasonStr(uint8_t code) {
    switch (code) {
        case 0: return "Not CFI";
        case 1: return "Syscall";
        case 2: return "Direct Conditional Jump Forward Taken";
        case 3: return "Return";
        case 4: return "Direct Call";
        case 5: return "Indirect Call";
        case 6: return "Direct Conditional Jump Forward Not Taken";
        case 7: return "Direct Conditional Jump Backward Taken";
        case 8: return "Direct Unconditional Jump";
        case 9: return "Indirect Conditional Jump";
        case 10: return "Indirect Unconditional Jump";
        case 11: return "Direct Conditional Jump Backward Not Taken";
        case 12: return "Max instructions reached";
        case 13: return "Max control-flow instructions reached";
        default: return "None";
    }
}

VOID InstrumentTrace(TRACE trace, VOID* v) {
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
```

```
        auto* bblInfo = new BBLInfo;
        allocatedBBLs.push_back(bblInfo);

        bblInfo->startAddress = INS_Address(BBL_InsHead(bbl));
        bblInfo->reasonCode = GetReasonCode(BBL_InsTail(bbl));
        bblInfo->instructionOffsets.reserve(BBL_NumIns(bbl));

        ADDRINT base = bblInfo->startAddress;
        for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins)) {
            ADDRINT delta = INS_Address(ins) - base;
            if (delta > 65535) {
                std::cerr << "Warning: offset too large for uint16_t, truncating";
            }
            uint16_t offset = static_cast<uint16_t>(delta);
            bblInfo->instructionOffsets.push_back(offset);
        }

        if (bblInfo->reasonCode == 6 || bblInfo->reasonCode == 7) {
            BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)LogBBL,
                           IARG_PTR, bblInfo,
                           IARG_BRANCH_TAKEN,
                           IARG_END);
        } else {
            BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)LogBBL,
                           IARG_PTR, bblInfo,
                           IARG_BOOL, FALSE,  // safe dummy value
                           IARG_END);
        }

    }
}

VOID Finish(INT32 code, VOID* v) {
    CommitTrace();
```

```
    std::ofstream csv(outputFileName);
    csv << "StartAddress,Instructions,CFICount,BranchOutcomes,Reason,Occurrences\n";

    for (const auto& entry : traceCache) {
        const Trace& trace = entry.first;
        csv << std::hex << "0x" << trace.getStartAddress() << std::dec << ","
            << static_cast<int>(trace.getInstructionCount()) << ","
            << static_cast<int>(trace.getCFICount()) << ","
            << std::hex << "0x" << static_cast<int>(trace.getBranchOutcomes()) << std::dec <<
","
            << ReasonStr(trace.reasonCode) << ","
            << static_cast<uint64_t>(entry.second) << "\n";
    }

    csv.close();
    for (auto* info : allocatedBBLs) delete info;
    allocatedBBLs.clear();
}

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

    traceCache.reserve(100000);
    if (PIN_Init(argc, argv)) return 1;

    maxInstructions = knobMaxInstructions;
    maxCFIs = knobmaxCFIs;
    startInstr = knobStartInstr;
    stopInstr = knobStopInstr;

    std::istringstream reasonStream(knobReasons.Value());
    std::string token;
    int index = 1;
    while (std::getline(reasonStream, token, ',') && index <= 11)
        reasonEnabled[index++] = (std::stoi(token) != 0);
```

```
    std::ostringstream nameStream;
    nameStream << "trace_results_" << maxInstructions << "ins_" << maxCFIs << "cfi_";
    for (int i = 1; i <= 11; i++) {
        nameStream << reasonEnabled[i];
    }

    nameStream << "_start_" << startInstr << "_stop_" << stopInstr;

    nameStream << ".csv";
    outputFileName = nameStream.str();

    TRACE_AddInstrumentFunction(InstrumentTrace, 0);
    PIN_AddFiniFunction(Finish, 0);
    PIN_StartProgram();
    return 0;
}
```

11 Appendix B

Trace.h

```
#ifndef TRACE_H
#define TRACE_H

#include <cstdint>
#include <functional>
#include <cassert>
#include <cstring>
#include "pin.H"

struct alignas(8) Trace {
    ADDRINT startAddress;        // 8 bytes
    uint32_t branchOutcomes;     // 4 bytes
    uint8_t instructionCount;    // 1 byte
    uint8_t cfiCount;            // 1 byte
    uint8_t reasonCode;          // 1 byte
    uint8_t _padding = 0;        // to fill to 16 bytes

    Trace()  :  startAddress(0),  branchOutcomes(0x1BADC0DE),  instructionCount(0),
cfiCount(0), reasonCode(0), _padding(0) {}

    inline void reset() noexcept {
        startAddress = 0;
        instructionCount = 0;
        cfiCount = 0;
        branchOutcomes = 0x1BADC0DE;
        reasonCode = 0;
        _padding = 0;
    }

    inline void recordFallthroughAddress(ADDRINT fallthroughAddr) noexcept {
        uint64_t hash = (fallthroughAddr ^ (fallthroughAddr >> 17) ^ (fallthroughAddr >> 34));
```

```
        branchOutcomes ^= static_cast<uint32_t>((hash * 0x9e3779b97f4a7c15ull) ^ (hash >>
33));
    }

    inline void incrementInstructions() noexcept { ++instructionCount; }

    inline ADDRINT getStartAddress() const { return startAddress; }
    inline uint8_t getInstructionCount() const { return instructionCount; }
    inline uint8_t getCFICount() const { return cfiCount; }
    inline uint32_t getBranchOutcomes() const { return branchOutcomes; }
    inline void setReasonCode(uint8_t code) noexcept { reasonCode = code; }
    inline uint8_t getReasonCode() const noexcept { return reasonCode; }

    inline bool operator==(const Trace& other) const noexcept {
        return startAddress == other.startAddress &&
               instructionCount == other.instructionCount &&
               cfiCount == other.cfiCount &&
               branchOutcomes == other.branchOutcomes &&
               reasonCode == other.reasonCode;
    }
};

static_assert(sizeof(Trace) == 16, "Trace struct size changed! Check packing or added
members.");

// Make sure std::hash<Trace> is defined properly BEFORE use
namespace std {
    template <>
    struct hash<Trace> {
        size_t operator()(const Trace& t) const noexcept {
            size_t h = std::hash<ADDRINT>{}(t.startAddress);
            h ^= static_cast<size_t>(t.instructionCount) * 0x9e3779b97f4a7c15;
            h ^= static_cast<size_t>(t.cfiCount) << 16;
            h ^= static_cast<size_t>(t.branchOutcomes) * 0x94d049bb133111eb;
```

```
            h ^= static_cast<size_t>(t.reasonCode) << 56;
            return h;
        }
    };
}

#endif // TRACE_H
```