UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Program Analyses that Scale to Large Codebases

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Min-Yih Hsu

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ardalan Amiri Sani
Professor Brian Demsky

2023

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

LIST OF ALGORITHMS **vi**

ACKNOWLEDGMENTS **vii**

VITA **viii**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Contributions . . . 5

2 Background **7**
- 2.1 Trade-off Between Performance and Precision . . . 8
- 2.2 Memory Aliasing . . . 11
- 2.3 Dataflow Analysis . . . 12
  - 2.3.1 Properties of transfer function . . . 14
- 2.4 Throughput Analysis . . . 15

3 Scaling Interprocedural Value-Flow Analysis to Large Codebases **17**
- 3.1 Introduction . . . 18
- 3.2 Background . . . 20
  - 3.2.1 Flow-Sensitive Value-Flow Analysis . . . 20
  - 3.2.2 Value-Flow Analysis as Graph Reachability . . . 22
  - 3.2.3 Reachability via Depth-First Tree Intervals . . . 23
- 3.3 DFI Design and Implementation . . . 25
  - 3.3.1 Preprocessing . . . 25
  - 3.3.2 Structure and Workflow . . . 28
  - 3.3.3 Intra-procedural Analysis . . . 29
  - 3.3.4 Inter-procedural Analysis . . . 35
- 3.4 Evaluation . . . 42
  - 3.4.1 Scalability of Different Client Analyses . . . 43

3.4.2 Comparison with PhASAR . . . 46
3.4.3 Size Distribution of Interval Set . . . 47
3.4.4 Precision . . . 48
3.5 Discussion . . . 50
3.5.1 Soundness and Comparison with IFDS . . . 50
3.5.2 Limitations . . . 53
3.6 Summary . . . 53

4 Efficient Whole-Program Throughput Estimation 54
4.1 Introduction . . . 55
4.2 Background and Motivation . . . 58
4.2.1 Static Throughput Estimation Challenges . . . 58
4.2.2 Differential Throughput Estimation . . . 63
4.3 Design . . . 64
4.3.1 Goals and Challenges . . . 64
4.3.2 Scalable Throughput Prediction . . . 66
4.3.3 Development-Driven Workflow . . . 67
4.3.4 Analysis Performance and Generalization Across Architectures . . . 67
4.3.5 Model Assumptions . . . 68
4.4 Implementation . . . 68
4.4.1 Instruction Broker . . . 69
4.4.2 Analysis Core . . . 70
4.4.3 Sub-Region Feature and Viewer Component . . . 73
4.5 Evaluation . . . 75
4.5.1 Differential Throughput Prediction . . . 75
4.5.2 Differential Throughput Prediction on Small Changes . . . 81
4.5.3 Scalability and Comparison . . . 83
4.6 Discussion . . . 85
4.7 Summary . . . 86

5 Related Works 88
5.1 Improvements on the Scalability of IFDS . . . 88
5.2 Sparse Value-Flow Analysis . . . 89
5.3 Program Analysis via Big Data analytics . . . 89
5.4 Static Timing Analyses . . . 89
5.5 Dynamic Timing Analyses . . . 91

6 Conclusion 93

Bibliography 95

LIST OF FIGURES

Page

2.1 Example C code for Section 2.1 9

3.1 A graph annotated with DFT intervals 24
3.2 Structure of the analysis engine in DFI 28
3.3 Caption for LOF 30
3.4 Caption for LOF 31
3.5 SSA def-use graphs for Listing 3.5 32
3.6 Non-tree edge handling 35
3.7 Caption for LOF 36
3.8 Interval sets breakdown for Listing 3.7. Values with italics names are reversed roots. 38
3.9 Scalability of time and memory consumption 45
3.10 Size distribution of interval sets 48

4.1 x86_64 assembly control flow. 59
4.2 High-level structure of MCAD 64
4.3 General workflow and interactions between the major components within MCAD. 69
4.4 Differential execution timing comparisons between subsequent development versions of `ffmpeg` and `clang`. Relative cycle count differences are plotted in blue and orange for MCAD and Perf respectively. The gray bars represent the Mean Absolute Percentage Error for each experiment, the geometric mean of all error values is plotted in red. In all experiments, we compare MCAD's predictions against hardware-performance counter measurements collected from execution traces on the respective physical devices. 77
4.5 Number of cache misses during the execution of different `ffmpeg` versions on CoffeeLake and Zen2 machines. 79
4.6 Differential throughput prediction of small software changes in `ffmpeg` and `Clang` on Intel CoffeeLake 82

LIST OF TABLES

Page

3.1 Percentage of SSA values with the respective number of uses 32
3.2 Description of analysis subjects we used in scalability evaluations 42
3.3 Performance of different client analyses in DFI and PhASAR. (#V+#E: Number of visited vertices and edges; # Q: Number of queries; AT: Analysis time; QT: Query time; TT: total time; Max RR: Max resident memory) 44
3.4 Precision of DFI and PhASAR on Coreutils programs (FN: false negatives; FP: false positives; TP: true positives) 51

4.1 # LoCC between clang release versions 80
4.2 # LoCC between clang minor release versions 81
4.3 # LoCC between ffmpeg release versions 81
4.4 # LoCC between ffmpeg minor release versions 81
4.5 Caption with FN 84

LIST OF ALGORITHMS

		Page
1	Interprocedural Worklist Algorithm	40

ACKNOWLEDGMENTS

First and for most, I would like to thank my advisor Prof. Michael Franz for giving me such an incredible opportunity to work with some of the best people in this field and giving tremendous amount of support through my entire PhD career. Michael always provides crucial steering and positive outlooks to the research direction while giving me lots of autonomy on exploring new ideas. He is also really supportive on the open source contributions I made as part of my research, which is an important mean to engage with broader audience in not just academia but also in the industry. Thank you, Michael.

I would like to express my gratitude to all the postdoctoral researchers I have been working with: Dr. Yeoul Na, Dr. Per Larsen, Dr. David Gens, Dr. Felicitas Hetzelt, and Dr. Adrian Dabroski. Thank you so much for your invaluable advice on new ideas and assistance to my research, especially on paper writing. I would also like to thank my fellow labmates and alumnus for their company and inspirations: Mitch, Fabian, Matt, Chinmay, Andre, Tommaso, Hongyu, Arthur, James, Paul, Prabhu, Joe, Stephan, and Dokyung. They make my graduate student life much more enjoyable.

To my internship mentors and managers: Vince & Stan (MediaTek), Josh (PlayStation), Adrian (Apple). Thank you for giving me a chance to improve my skills as a researcher and engineer in compiler development.

To the editors and publisher of my book *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries*, thank you for providing such an unique experience to create a book that both researchers and practitioners will find useful. I appreciate your time and patience during my writing while I was also studying in graduate school.

Finally, I would like to give special thanks to my committee members, Prof. Ardalan Amiri Sani and Prof. Brian Demsky, and my advancement committee members Prof. Joshua Garcia and Prof. Perry Johnson for their time and consideration.

This material is based upon work partially supported by United States Defense Advanced Research Projects Agency (DARPA) under contract W31P4Q-20-C-0052 and N66001-20-C-4027. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents or any other agency of the U.S. Government.

Last but not the least, I would like to thank my parents and my sister. I wouldn't be able to finish my degree without their moral support through all the ups and downs of my PhD life.

VITA

Min-Yih Hsu

EDUCATION

Doctor of Philosophy in Computer Science — **2023**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2022**
University of California, Irvine — *Irvine, California*

Bachelor of Science in Computer Sciences — **2018**
National Tsing Hua University — *Hsinchu, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2018–2023**
University of California, Irvine — *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant — **2019**
University of California, Irvine — *Irvine, California*

WORK EXPERIENCE

Compiler Engineer Intern — **2021**
Apple — *Cupertino, California*

Compiler Engineer Intern — **2020**
PlayStation (Sony Interactive Entertainment) — *San Mateo, California*

Compiler Engineer Intern — **2017, 2018, 2019**
MediaTek USA — *Woburn, Massachusetts*

REFEREED CONFERENCE PUBLICATIONS

Min-Yih Hsu, Felicitas Hetzelt*, David Gens*, Michael Maitland, and Michael Franz. "A Highly Scalable, Hybrid, Cross-Platform Timing Analysis Framework Providing Accurate Differential Throughput Estimation via Instruction-Level Tracing" In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* 2023
* Equal Contribution Authors

ABSTRACT OF THE DISSERTATION

Efficient Program Analyses that Scale to Large Codebases

By

Min-Yih Hsu

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Michael Franz, Chair

Program analysis extracts software properties that are helpful to developers and provides invaluable insights into a program; it is essential to many computer science areas such as compiler optimizations, cybersecurity, and performance engineering, to name a few. In the past few decades, however, researchers and practitioners have found difficulties in scaling up program analyses of all kinds, both static or dynamic ones, with modern software whose size and complexity have rapidly increased. In this dissertation, we focus on two of the most important program analysis areas, dataflow and throughput analysis, and create two frameworks – DFI and MCAD, respectively – with the principle of combining several optimizations in a novel way to overcome the scalability issues in their areas.

Our contributions in DFI augments IFDS, an efficient dataflow analysis framework for distributive dataflow problems, with a more efficient algorithm and a sparse program representation. Our key insight is an entirely new way of looking at IFDS problems and the realization that doing things in the *reverse* direction of the graph reachability algorithm leads to much better performance; scaling to programs that were previously out of reach due to physical memory constraints.

MCAD is a novel way of predicting the performance characteristics (*e.g.* total cycle counts and instructions per cycle) of a program, based on combining the best ideas from both

static and dynamic throughput analysis. It uses dynamic traces and run time information to drive a static throughput analysis engine. Such combination avoids poor accuracy across branch boundaries traditionally suffered by static approach and more importantly, has much better scalability than dynamic methods like cycle-accurate emulators. MCAD also shows comparable accuracy in differential throughput analysis.

Together with other works in the fields, this dissertation provides building blocks and novel ideas of combining different techniques for program analyses to scale up with the ever-growing size, complexity, and challenges of future software codebases.

Chapter 1

Introduction

1.1 Overview

Program analysis has been the stepping stone of computer science since its dawn. It constitutes a huge family of techniques that inspect, comprehend, or dissect programs, in pursue of key properties that further enable an even larger body of applications; playing crucial roles in areas like compiler optimization, bug detection, security enhancement, and performance engineering, to name a few. However, program analysis faces great challenges to modern software projects as it struggles to scaling up with the size and complexity of their gigantic code bases. Imposing great overhead on both analysis performance and space (*e.g.* memory and storage) consumption. Such complication forces many users to make compromise on the coverage and effectiveness brought by program analysis, or even hinders the willingness to perform any analysis at all.

A program analysis can be described in two broad categories: static and dynamic. Static analysis processes the program, normally ahead of time, without any concrete execution. Since the exact program paths are unknown during the analysis, approximations are required

to yield a safe result that satisfies all possible program states. Analyses fall into this category are usually used to provide estimations on the target programs with a holistic view. But they come with a cost of inferior precision due to their over-approximating and conservative nature.

A typical example of static analysis is dataflow analysis. Dataflow analysis is designed to extract key program properties for usages in compiler optimization, security analysis, and bug finding, to name a few. It models program properties as abstract facts and program statements as transfer functions over these facts. Traditionally, a dataflow analysis iteratively applies these transfer functions along the control flow until the result reaches a fixed point. However, such model is inefficient because it easily produces lots of unnecessary information that is never used [38]; scaling to inter-procedural code is more challenging, too.

Trading off precision for performance is one of the most common ways to solve the aforementioned problem. For instance, in a context-*insensitive* analysis, function calls are generally ignored; a field-*insensitive* analysis might not be able to distinguish field accesses on the same object. In addition to operating with a reduced precision, limiting the capability of a dataflow analysis to a specific class of problem is also commonly used. For example, Reps et al. [51] pioneered an efficient way to solve dataflow problems that can be modeled with a powerset of atomic dataflow facts and distributive transfer function, also know as the IFDS problems. Specifically, IFDS problems can be precisely solved with a graph reachability algorithm. This convenient property can also be leveraged in an inter-procedural analysis with summary-based approach. Summary-based approach abstracts functions into summaries that can be plugged into their respective call sites to be co-analyzed in the caller's context. Despite being extremely efficient, such solving technique reduces precision in most classes of problems – except those using distributive transfer function, which will not suffer any precision degradation.

In recent years, people have also been exploring different kinds of *sparse* dataflow analysis.

The idea of sparse analysis is using an efficient program representation to capture only the necessary part of the program and potential dataflow facts. For instance, for a points-to analysis, modeling pointers and their flow among memory operations is usually sufficient. These type of program representations are usually organized as a graph: Single-Assignment (SSA) graph, Value-Flow Graph (VFG), or Sparse Expression Graph (SEG) are some of the most commonly used formats. As opposed to following the control flow, sparse analyses traverse along the vertices and edges in such graphs, in order to reduce the number of redundant traversals during the analysis.

In Chapter 3, we propose a new value-flow analysis framework, combining the aforementioned scalability improvements, called DFI, which solves IFDS problems on SSA form programs in several novel ways. Key to our approach is a novel interval-based graph reachability algorithm that performs depth-first traversal in a reverse direction, yielding a low tree width version of the same SSA graph, which is already a sparse graph representation for value flows. This gives us the ability to solve graph reachability between SSA values with much better resource efficiency and performance characteristics than previous approaches and provide almost linear scalability to truly large programs.

A dynamic program analysis makes observations on the concrete execution of target programs. It extracts desired properties directly from the program states during run time. For instance, variable values, exceptions, processor states, and most importantly, the program paths. Knowing the exact execution path helps dynamic analyses to yield precise results, as opposed to the over-approximations used by its static counterparts. However, dynamic analyses are known to suffer from the coverage problem. Specifically, it's difficult to generate proper inputs that trigger every possible program paths. Leading to incomplete or even unsound analysis results.

Dynamic program analysis is often used in performance engineering like throughput analysis. Throughput analysis aims to estimate the cycle counts and instruction per cycle (IPC) of

a specific part of program. It is crucial to the development workflow of many performance sensitive applications. Modern processors employee a large number of optimizations, like superscalar architecture and out-of-order execution, to increase the program throughput without any code modification. However, the exact execution model of such optimizations is usually proprietary, varying between different hardware vendors or even processor models, and hard to predict key throughput properties like instruction latency and number of cache misses. Making timing estimation on the target programs more difficult.

Traditionally, cycle-accurate emulator has been a popular dynamic approach to the aforementioned problems. Tools fall under this category faithfully model the architectural details to match processor's run time behavior, providing concrete and generally accurate estimates. Nevertheless, they suffer from high analytic performance overhead and can require hours to days for analyzing a program. Furthermore, architectural simulators exhibit a high architecture dependence and are complicated to set up, often requiring dedicated expert knowledge and/or giving rise to compatibility issues with standard tools and default environments.

In Chapter 4, we present MCAD, a lightweight alternative that provides whole-program throughput prediction of binary software. MCAD follows a hybrid approach that supplements static throughput estimates with dynamic runtime information. To avoid the overhead of cycle accurate architectural simulation, MCAD uses an emulation-based approach to obtain execution traces using QEMU [12] and dynamically forwards them to the LLVM Machine Code Analyzer (MCA) [2] for instruction-level analysis. MCAD improves on the state of the art by scaling up to complex real-world software. It is well suited to providing cycle count estimates with rapid developer-centric turn-around times while targeting a range of different hardware architectures.

1.2 Contributions

This dissertation makes the following contributions, organized by chapter:

Chapter 3

- **Efficient Graph Reachability**. A novel tree like graph structure that encodes inter-procedural, flow-sensitive, context-sensitive and sparse value flow information and can resolve reachability queries in near constant time,
- **Linear Scalability**. An algorithm to construct the value flow graph whose memory and runtime requirements scale almost linearly with the program size, and
- **Open Source Implementaiton**. A full source-language agnostic, LLVM IR-based implementation of our technique that scales to large real-world projects containing hundreds of thousands of lines of code.

Chapter 4

- **Novel Differential Timing Analysis**. We present MCAD: a new open-source framework for throughput estimation yielding highly accurate differential timings, on par or better than the current state-of-the-art, while reducing turnaround time by several orders of magnitude.
- **Hybrid Approach**. Our prototype implementation leverages QEMU as a fast instruction executor, utilizing MCA to model individual per-instruction execution cycles, rather than simulation-based approaches that faithfully model complex processor front-ends.

- **Efficient and Accurate Analysis on Real-World Cases**. We provide a detailed evaluation of MCAD with respect to accuracy and scalability for the popular x86 and ARMv8 instruction-set architectures using several different devices and hardware-performance counters to collect timing measurements for real-world traces as ground truth.

Chapter 2

Background

Program analysis studies the properties of software artifacts. It provides a better comprehension on the target software that can be further used in a wider variety of goals and applications, ranging from performance improvements to security enhancements. The complexities of those tasks usually grow up with the scale of the code bases. However, with the increasing number of code bases exceeding millions lines of code, researchers and practitioners have been struggling to keep the time and resources spending on program analyses reasonable.

We focus on the scalability issues in two of the most prominent program analysis areas: performance estimation and dataflow analysis. To familiarize the readers with these two program analysis areas, we first cover some concepts shared between them in Section 2.1 and Section 2.2. Then, Section 2.3 and Section 2.4 provide introductions to the foundations of their respective discussions in Chapter 3 and Chapter 4, in which additional area-specific background will be provided in the beginning of each chapter.

2.1 Trade-off Between Performance and Precision

When designing a program analysis – either a static or dynamic one, in many cases people have to make trade offs between *performance* and *precision*, which are properties that usually trend inversely proportional to one another. Finding the sweet spot between them for a specific application has always been a hard problem in this area.

The performance of a program analysis is usually described in more straightforward ways, including its run time, space consumption (*i.e.* memory and storage space) and the size of programs under analysis. On the other hand, the precision of a program analysis is usually characterized by its **sensitivities**. There are primarily four kinds of sensitivities, differing from each other on the resolution, in the form of program constructions or concepts, of which their analysis results can yield:

- Flow sensitivity. Whether the analysis considers *control flows*.
- Context sensitivity. Whether the analysis considers (inter-procedural) function calls, specifically the *caller contexts*.
- Field sensitivity. Whether the analysis can distinguish individual fields in an aggregate-type value, like a struct.
- Path sensitivity. Similar to flow-sensitivity, but it can distinguish not just the general control flow but also individual program path.

Here, we are particular interested in the first two sensitivities for both static and dynamic program analyses. These sensitivities will be covered in detail in the following paragraphs, along with their potential impacts on the analysis performance.

To motivate our discussions, we perform a taint analysis on the C snippet depicted in Fig-

ure 2.1. In this analysis, the only source of taint is the return value of function `taint_src` (line 6); the analysis reports a positive result if any tainted value can reach the first argument of function `taint_sink` (line 7).

```
1  struct Agg {
2    int f1;
3    int f2;
4  };
5
6  int taint_src();
7  void taint_sink(int v);
8
9  void foo(struct Agg *x) {
10   int v = 0;
11   taint_sink(v);
12   if (x)
13     v = x->f1;
14   taint_sink(v);
15 }
17 void bar() {
18   struct Agg obj;
19   obj.f2 = taint_src();
20   foo(&obj);
21 }
22
23 void zot() {
24   foo(NULL);
25 }
```

Figure 2.1: Example C code for Section 2.1

Flow sensitivity A flow sensitive analysis takes control flow locations into considerations. Take function `foo` in Figure 2.1 as an example, there are two calls to `taint_sink` on line 11 and 14; both consume variable `v`. A flow-*insensitive* analysis always yields a positive result regardless of which `taint_sink` call sites the users are asking, as long as variable `v` is tainted *somewhere* in the entire program. This leads to false positive at line 11, as it consumes nothing but a (non-tainted) constant so it will never be tainted. On the other hand, a flow-sensitive analysis can correctly tell that only line 14 is considered a potential tainted site.

In principle, such sensitivity is achieved by tracking the variable in interest (in this case `v`) through control flow path. Traditionally, this is done by maintaining certain data structures at *each* program point. For instance, in this case, both dynamic and static analyzers can maintain a set of tainted variables before (or after) each statement [30]. However, such

per-statement data structures don't usually scale well with the number of statements as well as tracked variables, in terms of run time and memory consumption. As they might grow quadratic or even cubic [51, 57, 28] with the aforementioned factors in the worst case. More importantly, many of the collected information on each statement might turn out be either redundant or unused [38], hence a waste of efforts.

Flow-insensitive analyses tend to use a single data structure to model the analysis problem of a sub-program construction (*e.g.* function) or the entire program. Abstracting away all the control flow details. For instance, they can use a single set to carry all possible tainted variables in Figure 2.1. As a consequence, they usually scale better with program complexity as well as the analysis domain. Of course, this approach – especially when using with static analyzers – can easily create false positives due to its inability to distinguish different control flow points, as we illustrated earlier with variable v in Figure 2.1.

Context sensitivity A context sensitive analysis is able to calculate the results of a function based on its caller contexts. Using function `foo` in Figure 2.1 as the example again, there are two callers in the same snippet: `bar` and `zot`. Since `zot` always calls `foo` with a `NULL` value, line 12 will never be taken, hence both line 11 and 14 will never be tainted under this calling context. Nevertheless, a context-insensitive analysis is unable to make such inference and yield a false positive analysis result for `foo` function called at line 24, which is something that does not happen for a context-sensitive analysis.

There are several ways to implement a context-sensitive analysis. For a static analysis the most straight forward way is by inlining every callee functions to their call sites. This essentially turns an inter-procedural analysis problem into an intra-procedrual one, bringing the benefits of little or no modification to the original analysis algorithm. However, such approach can easily increase memory pressure as the program under analysis bloats. Moreover, it's tricky to handle recursion and an analysis usually sets an upper bound on the number

of recursive levels it will track down [68].

Another common approach taken by context-sensitive static analyses is using function summary. Specifically, the analysis summarizes the ingress and egress information of a callee function, for each caller context, and inserts such summary into the respective call site. This method avoids the aforementioned code bloating issue. Nevertheless, it still imposes limitations on recursion, which is usually mitigated by over-approximation with lost of precision. Such precision lost can be avoided if the analysis has certain data flow property, specifically a distributive transfer function, which will be covered later in Section 2.3 and Chapter 3.

2.2 Memory Aliasing

The problem of memory aliasing in program analysis stems from indirect memory operations, where values are accessed through memory addresses known during run time. Two memory addresses are **aliases** if the memory objects they point to are overlapped, either partially or completely. Memory aliasing can have significant implications on program behavior, including the potential for unexpected side effects, bugs, and performance issues.

One of the primary challenges associated with memory aliasing is that it can lead to unintended modifications of data. When multiple pointers or references are allowed to access and modify the same memory location, changes made through one pointer can affect the values accessed through other pointers. This can lead to incorrect program behavior, as it violates the assumption that each memory location is only modified by a single entity. Memory aliasing-related bugs can be challenging to diagnose and fix, as the root cause may be distant from the observable symptoms.

Memory aliasing also poses challenges to both static and dynamic program analyses. For instance, when a compiler encounters memory aliasing, it needs to consider all possible paths

through the program to determine if a memory location can potentially be modified. This can limit the ability of the compiler to apply more aggressive optimizations such as loop unrolling, common subexpression elimination, or instruction scheduling. The presence of memory aliasing can result in conservative assumptions by the compiler, leading to suboptimal generated code and reduced performance.

Out-of-order execution is an optimization technique commonly seen in modern processors. It reorders incoming instruction sequence to increase throughput and maximize the degree of Instruction Level Parallelism (ILP). In order to preserve the semantic of the original program, online dependency analysis plays an important part in this process, which boils down to resolving dependencies on resources like register files and, more importantly, accessed memory addresses. It has to make sure instructions that accesses memory aliases can't be reordered. This is a complicated dynamic analysis for both the hardware and various emulators used in both functional and cycle-accurate program analysis. Emulators of such kind usually take weeks or even months on running larger programs due to the aforementioned complexity, making scalability one of the most prominent issues in its field.

2.3 Dataflow Analysis

Dataflow analysis is the bread and butter for performing static program analysis. It helps users to get an approximation of data movements during run time in a static fashion. This data is usually abstract program facts or properties in which users are interested in. For instance, memory objects pointed by a certain pointer, constant values, and tainted variables, to name a few. A dataflow analysis usually evaluates such program facts at each statement following the control flow order, going either *forward* or *backward*. We can express a forward

dataflow analysis with the following model:

$$OUT_s = GEN_s \cup (IN_s - KILL_s)$$

$$IN_s = \underset{\forall i \in pred(s)}{\sqcap} OUT_i$$

In this formulation, IN_s and OUT_s are the input and output sets of dataflow facts for statement s, respectively; GEN_s and $KILL_s$ are the respective sets of facts generated and killed by s; $pred(s)$ is the set of predecessors of s, while $succ(s)$ is its successors counterpart; $\sqcap$ is the meet operator we will cover shortly. Using taint analysis as an example, which tries to find the set of variables that are tainted at a given program point, GEN_s is the set of variables newly tainted by s; $KILL_s$ can be the set of "sanitized" variables.

We can further rewrite the above formulation into:

$$f_s(IN_s) = GEN_s \cup (IN_s - KILL_s)$$

$$IN_s = \underset{\forall i \in pred(s)}{\sqcap} f_i(IN_i)$$

Where we abstract each statement s into a *transfer function* f_s over the inputs IN_s.

Determining program executions statically is difficult, as this problem can be boiled down to the halting problem [59], which is famously undecidable in the general case. Thus, people usually design their analyses in a way that *over-approximates* the real executions that accounts for every possible program states. The key to such over-approximation is the assurance on soundness (*i.e.* does not yield incorrect results). To this end, dataflow facts are usually modeled using **lattice**. A lattice L is partially ordered set in which a partial order $\sqsubseteq$ is defined between some of the elements. In the context of dataflow analysis, each element in L is modeled as a possible dataflow result. Given two elements $a, b \in L$, $a \sqsubseteq b$ can be interpreted as "a is a *safer* result than b".

A safe dataflow result can be used by transformations or optimizations without breaking the original program semantics. The exact interpretation of "safe" and definition of $\sqsubseteq$ vary among different analyses. For instance, to a taint analysis it is always considered safe to claim that all variables are tainted, albeit being extremely imprecise. Thus, $a \sqsubseteq b$ is defined using superset operator $a \supseteq b$ for $a, b \in L_{taint}$. In contrary, to an available expression analysis, which gives a set of expressions that need *not* to be recomputed at each program point, it is considered safe to make every expressions subject to recomputation. Therefore, $a \sqsubseteq b$ is defined using *subset* operator $a \subseteq b$ for $a, b \in L_{avlexpr}$.

When we consider a statement with multiple predecessors, like joining two branches, we have to specify a way to merge dataflow facts coming from different predecessors with meet operator $\sqcap$. Similar to $\sqsubseteq$, The exact definition of $\sqcap$ depends on the analysis. For instance, in taint analysis, $\sqcap$ is a set union. On the other hand, in an available expression analysis $\sqcap$ is a set *intersection* on invariant expressions from all predecessors.

2.3.1 Properties of transfer function

A dataflow analysis usually traverses along control flow to evaluate the desired dataflow facts on each statement. In the ideal model, this is done by enumerating every possible program paths and apply transfer functions along the path accordingly, before merging these path-based results at the end. Such solution is also called **Meet-Over-Paths (MOP)**. However, MOP is generally computational *infeasible* due to the inherent path explosion problem in static analysis. In practice, people usually prefer another kind of solution called **Maximum Fixed Point (MFP)**. The idea of MFP is to merge dataflow facts whenever the control flows join, in conjunction with an iterative procedure that repeats until a fixed point solution is found. The seminal work by Kam et al. [30] showed that if the transfer function is *monotone*, such fixed point solution can be found.

Given a finite lattice L, partial order $\sqsubseteq$ and meet operator $\sqcap$, a transfer function f is said to be **monotonic** iff:

$$f(a \sqcap b) \sqsubseteq f(a) \sqcap f(b) \text{ where } a, b \in L$$

The monotonic property effectively ensures that dataflow facts during the iterative process only gradually become safer. More formally speaking, the lattice elements of dataflow facts only progress in a single direction in the lattice. And since the lattice has a finite height, this process is guarantee to terminate.

Similar to monotonic function, a transfer function is said to be **distributive** iff:

$$f(a \sqcap b) = f(a) \sqcap f(b) \text{ where } a, b \in L$$

Transfer functions with this property hold the same precision regardless of applying them before or after a merge. It is a subset of monotonic function.

2.4 Throughput Analysis

In the field of performance engineering for computer programs, throughput analysis plays a critical role in assessing and optimizing the efficiency and scalability of software applications. Throughput analysis focuses on measuring the rate at which a program can process a specific workload or perform a set of tasks. It helps software developers and performance engineers identify performance bottlenecks, understand system limitations, and improve the overall throughput of their applications.

One of the primary objectives of throughput analysis in performance engineering is to identify and address performance bottlenecks within a computer program. Performance bottlenecks are points in the code where the program's execution slows down or becomes inefficient,

limiting the overall throughput of the application. By analyzing the flow of work, examining resource utilization, and measuring metrics such as response time and throughput rate, performance engineers can pinpoint the areas that negatively impact performance. Throughput analysis helps identify inefficient algorithms, resource-intensive operations, or synchronization issues, allowing developers to optimize code and improve the throughput of the program.

Furthermore, throughput analysis is crucial for evaluating the scalability of computer programs. Scalability refers to the ability of a program to handle an increasing workload or user demand without a significant decrease in performance. Throughput analysis helps performance engineers assess how the program's throughput changes as the workload or the number of concurrent users increases. By conducting load testing and analyzing performance metrics under different workloads, engineers can identify scalability limitations, determine the maximum throughput the program can handle, and make necessary optimizations to ensure the program can scale effectively.

Moreover, throughput analysis provides valuable insights into the impact of system configuration and hardware resources on program performance. By varying parameters such as the number of processing cores, memory allocation, or disk I/O rates, performance engineers can analyze how different system configurations affect the program's throughput. This information enables engineers to optimize resource allocation, fine-tune system settings, and select appropriate hardware components to maximize program throughput and achieve optimal performance.

Throughput analysis is a crucial aspect of performance engineering for computer programs. By identifying performance bottlenecks, evaluating scalability, and analyzing the impact of system configuration, performance engineers can optimize code, allocate resources effectively, and improve the overall throughput and performance of software applications. Through this analysis, developers can ensure their programs can handle increasing workloads and meet the performance expectations of end-users.

Chapter 3

Scaling Interprocedural Value-Flow Analysis to Large Codebases

Context- and flow-sensitive value-flow information is an important building block for many static analysis tools. Unfortunately, current approaches to compute value-flows do not scale to large codebases, due to high memory and runtime requirements. This paper proposes a new scalable approach to compute value-flows via graph reachability. To this end, we develop a new graph structure to represent instruction dependencies which (i) is highly time and space efficient in its construction, and (ii) allows us to resolve reachability queries in near constant time. Underlying our approach are two key insights. First, the analysis domain is restricted to represent only sparse SSA def-use chains and second, the graph construction traverses def-use chains in opposite direction. Traditionally, dependency graphs are constructed in forward direction resulting in graphs with a high number of non-tree edges. Our approach instead minimizes the tree width of the resulting graph which allows us to employ time and space efficient tree traversal algorithms to compute graph reachability and at the same time enables efficient graph computation and storage.

We present a value-flow analysis framework, DFI, implementing our approach. We compare DFI against a state-of-the-art value-flow analysis framework, PhASAR, to extract value-flows from 10 real-world software projects. Compared to PhASAR, DFI reduces analysis time by a factor of 14 and at the same time reduces memory requirements by a factor of 3 on average. Our analysis shows that, in contrast to previous approaches, DFI's memory and runtime requirements scale almost linearly with the number of analyzed instructions.

3.1 Introduction

Value-flow analysis is a subset of dataflow analysis that helps to statically reason about the dependencies among program constructs such as variables and memory blocks. It underpins many crucial program analysis techniques that are widely used in compiler optimizations and for finding security vulnerabilities: taint analysis [24, 10], uninitialized variable analysis [63], live variable analysis [36], and available expression analysis, to name a few. In order to operate effectively, such techniques usually require precise value-flow information that is context- and flow-sensitive as well as interprocedural [62].

Seminal work by Reps et al. [51] demonstrated that interprocedural context- and flow-sensitive analysis for *finite, distributive, subset* value-flow problems can be expressed as reachability between nodes within a program graph. Their algorithmic framework *IFDS* operates on a graph structure in which each node maps to a value-flow fact within the target program. But, while solved in theory, in practice the adoption of static value-flow frameworks is still limited by severe scalability issues. In fact, as we will show in Section 3.4, currently there exists no static analysis framework that is able to solve such kind of value-flow problems for larger real world codebases such as OpenSSL and FFmpeg on single commodity PCs.

The memory and runtime requirements of static value-flow analysis frameworks such as IFDS are largely governed by the graph representation of the program and the performance of the graph reachability algorithm. Indeed, previous approaches to solving the scalability problem have often focused on working around the shortcomings of established graph-reachability algorithms [28, 37, 9]. In contrast, in this paper, we present a solution that is based on a novel sparse graph representation, leading to a reachability algorithm that is highly efficient in answering value-flow queries.

Our key insight is that by restricting value-flow facts to SSA values, the program graphs representing value-flow can be constructed to have a low tree width [52], which is a measure of how similar a graph is to a tree. Specifically, we found that performing a depth-first traversal in the *opposite* direction of SSA def-use chains will result in a graph with a significantly reduced number of *non-tree* edges. Based on this insight, we develop a novel graph reachability algorithm based on tree traversal which significantly reduces processing time and memory requirements compared to previous approaches. In fact, our new algorithm allows us to determine dependencies between two *arbitrary* instructions in near constant time for most queries. Additionally, the resulting graph representation is *sparse*, thereby further reducing processing time and memory requirements by incorporating only program statements relevant for value-flow propagation.

We implemented these ideas in a framework that we call DFI[1]. DFI is an interprocedural flow- and context-sensitive value-flow analysis framework for a specific kind of finite, distributive, subset problems with SSA values as their value-flow facts. To quantify the improved performance of our approach, we evaluate DFI against PhASAR [53], a state-of-the-art static dataflow analysis framework for LLVM IR. PhASAR is a popular IFDS-based framework that can solve the same kind of value-flow problems as DFI. As our evaluation shows, DFI is able to scale to significantly larger codebases on commodity hardware than PhASAR, using

[1]The name DFI pays homage to IFDS, while our reversal of the letters alludes to the fact that our technique processes nodes in the reverse order.

significantly less memory overall, and running significantly faster. Therefore, DFI constitutes the first static solution to resolve interprocedural context- and flow-sensitive value-flow for real-world software projects, reaching to over a million lines of code, under realistic resource limitations. In summary our contributions are the following:

- a novel tree like graph structure that encodes interprocedural, flow-sensitive, context-sensitive and sparse value flow information and can resolve reachability queries in near constant time,
- an algorithm to construct the value flow graph whose memory and runtime requirements scale almost linearly with the program size, and
- a full source-language agnostic, LLVM IR-based implementation of our technique that scales to large real-world projects containing hundreds of thousands of lines of code.

We further pledge to make the source code of our project freely available under an open-source license.

3.2 Background

In this Section, we explain terminologies and concepts that will serve as the building blocks for rest of the paper.

3.2.1 Flow-Sensitive Value-Flow Analysis

Value-flow analysis models the specific propagation of data through a program's storage locations. A flow-sensitive analysis respects the program's control flow and calculates results for each program point. In contrast, flow-insensitive analysis ignores statement ordering and

computes a single solution that is sound for all program points. Traditionally, to achieve flow-sensitivity, value-flow analysis is built on top of a monotone dataflow analysis framework [30]. The analysis follows control flows in the control-flow graph (CFG), while accounting for changes in dataflow facts made by each statement. For example, in a taint analysis, dataflow facts are represented by a set of tainted variables at a given program point. For each statement i, the transfer function calculates two sets: IN_i and OUT_i, which represent the sets of dataflow facts that hold right before and right after statement i. In the context of a taint analysis, the transfer function might add (or subtract) variables that are (or not) tainted. IN_i is equal to OUT_{i-1} passed down from previous statement in the control flow, in the case of having multiple predecessor statements, IN_i is the set union of $OUT_p, \forall p \in predecessors(i)$. The algorithm repeatedly performs such updates on every statement until it reaches a global fixed point.

This approach usually comes with a high performance overhead. Dataflow facts are propagated to every program point, despite the fact that only a small portion of them is needed to answer the dataflow query we are interested in. In addition, maintaining two sets per statement tends to create a large memory footprint as well.

SSA-Based Sparse Value-Flow Analysis

In recent years, sparse value-flow analysis [25, 57, 48, 26] has introduced a promising solution to the aforementioned problem. Sparse value-flow analysis avoids propagating dataflow facts through program statements that are unrelated to the analysis. A common way is to perform the analysis on Static-Single Assignment (SSA) form [21] of programs.

In SSA, each variable is defined exactly once in a "static" view of the program, i.e., disregarding "dynamic" reassignments of variables that may happen at the same program location inside of loops. If there are multiple static definitions of a variable in the original program (such as multiple separate program locations that perform assignments to the variable), then

each such assignment of the original program variable becomes a separate SSA variable or value. SSA form expresses value definitions and usage information, also called def-use chains, explicitly, so that tracing values to their immediate uses becomes much easier. Therefore, def-use chains can help us to rule out irrelevant data flows and greatly improve the efficiency of value-flow analyses [49].

Tracking every variable in SSA form is difficult as it has to account for potential aliasing among pointer variables. To tackle this problem, mainstream compilers such as GCC [46] and LLVM [33] adopt a variant of SSA called *partial* SSA which divides variables into two categories: top-level and address-taken. Top-level variables cannot be referenced indirectly via a pointer and can be trivially converted into SSA form; address-taken variables are referenced indirectly via top-level pointer variables and are not represented in SSA form at all. This additional layer of indirection makes it more difficult for us to track value-flows on address-taken variables in partial SSA. In Section 3.3.1, we will show our solution to this problem.

3.2.2 Value-Flow Analysis as Graph Reachability

Reps et al. pioneered the idea of solving traditional dataflow analyses by turning them into graph reachability problems in their IFDS framework [51], inspiring a whole body of research on graph reachability based program analysis [60, 31, 11, 42]. IFDS is designed to solve finite, distributive, subset dataflow problems. These problems have distributive transfer functions defined as $F \subseteq 2^D \rightarrow 2^D$, where D is a finite set of dataflow facts. In other words, the problem domain is restricted to the powersets of dataflow facts. IFDS can thereby describe local transfer functions on each statement as a mapping between dataflow facts before and after the statement. The transfer functions connect individual dataflow facts across statements via edges that transitively compose into paths in a larger *exploded*

supergraph. An intraprocedural dataflow query then boils down to computing whether there is a valid path between the vertex representing the dataflow source and the program points we are interested. Interprocedurally, the paths are extended from a call site to a callee and back to the same call site. Context-sensitivity is achieved by matching call and return edges.

The IFDS framework can operate on arbitrary domains D given that they are finite. In contrast, DFI restricts the dataflow fact domain to SSA values covering a wide range of crucial value-flow analyses like taint analysis and uninitialized variable analysis, to name a few.

3.2.3 Reachability via Depth-First Tree Intervals

A *Depth-First Tree (DFT)* is a common approach to compute graph reachability. A DFT is an ordered spanning tree derived from the process of *Depth-First Search (DFS)* [20]. Each vertex of a DFT is assigned an integer interval $\langle s_v, e_v \rangle$: s_v is the discovery timestamp when v is first visited and e_v is the finish timestamp when all out-neighbors of v have been visited. The timestamp is initialized with zero and it is increased by one upon visiting a new out-neighbor from a vertex. Formally speaking, an interval has the following invariant:

$$s_v, e_v \in \mathbb{Z}, s_v \geq 0 \wedge e_v \geq 0 \wedge e_v > s_v$$

Figure 3.1 shows an example graph with its DFT intervals. To simplify the problem without losing generality, we always add a pseudo vertex δ to connect every vertex in the graph, such that the spanning tree has a single root.

For a given vertex in the spanning tree, its interval always *subsumes* the intervals of all vertices in its subtree. An interval $\langle s_k, e_k \rangle$ is said to subsume another interval $\langle s_l, e_l \rangle$, denoted

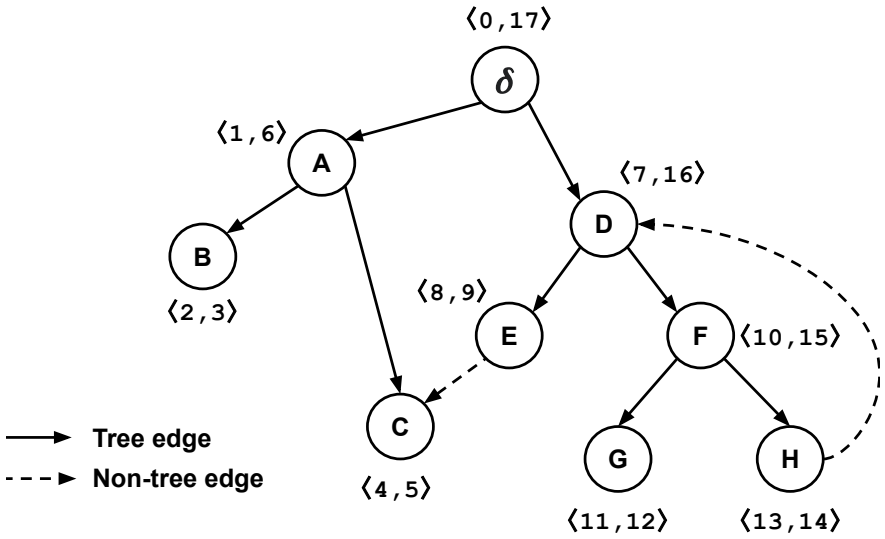


Figure 3.1: A graph annotated with DFT intervals

by $\langle s_k, e_k \rangle \supseteq \langle s_l, e_l \rangle$, if they have the following properties:

$$s_k \leq s_l \wedge e_k \geq e_l \implies \langle s_k, e_k \rangle \supseteq \langle s_l, e_l \rangle$$

Vertex F in Figure 3.1, for example, has an interval of $\langle 10, 15 \rangle$, which subsumes the intervals of its children $\langle 11, 12 \rangle$ and $\langle 13, 14 \rangle$. Conversely, vertex E and F are siblings so none of their intervals subsumes one or the other. We distinguish between tree edges (solid lines) and non-tree edges (dashed lines) in Figure 3.1. There are two types of non-tree edges: cross edge and back edge. They are defined using the interval relationship: for vertices k and l, alone with their corresponding intervals $\langle s_k, e_k \rangle$ and $\langle s_l, e_l \rangle$, Edge $k \to l$ is

- a cross edge if $e_l < s_k \vee e_k < s_l$, and
- a back edge if $\langle s_l, e_l \rangle \supseteq \langle s_k, e_k \rangle$.

For example, in Figure 3.1, edge $E \to C$ is a cross edge, and edge $H \to D$ is a back edge.

By comparing the intervals of arbitrary two vertices, using the subsuming relation defined

earlier, we can easily know whether they can reach each other through (spanning) tree edges in *constant time*. However, this property only holds for tree edges. For reachability queries on generic graphs, we also need to consider paths that go through non-tree edges. To answer those queries, several previous works [61, 64, 29] have proposed a variety of ad-hoc solutions on top of DFT intervals. In Section 3.3.3, we are going to introduce an augmented DFT-interval graph reachability algorithm to solve this problem.

3.3 DFI Design and Implementation

DFI is designed to be programming-language agnostic on its input program and can process any program compiled into LLVM IR. Section 3.3.1 will cover necessary preprocessing to convert this initial LLVM IR into a form more favorable to value-flow analysis. Sections 3.3.2 to 3.3.4 will cover the details of our main algorithm and implementation.

3.3.1 Preprocessing

DFI relies heavily on the sparse SSA program representation to track value-flows efficiently. However, as discussed in Section 3.2.1, the partial SSA form used by LLVM IR excludes address-taken variables. To track the value-flows of address-taken variables we convert the original input program into a custom representation that improves handling of indirect memory operations and pointers.

We implement this custom program representation using MLIR [34], a versatile framework to create custom intermediate representations (IRs) for program analyses and compiler optimizations. A custom IR is called *dialect* in MLIR. It consists of a type system and various building blocks that define the semantics like operation, block, region, and attribute, to name a few. A program statement or LLVM IR instruction is usually modeled by a MLIR

operation. We create our own DFI dialect on top of existing LLVM IR constructions with three custom operations: `dfi.store`, `dfi.may_use` and `dfi.call`.

`dfi.store` and `dfi.may_use` operations

To make value flows on address-taken variables explicit in SSA from, DFI introduces two new memory operations to the dialect: `dfi.store` and `dfi.may_use`. Inspired by the idea of MemorySSA [19, 47]. `dfi.store` acts like a normal `llvm.store` instruction with two additional features in its IR form: (i) it can carry an auxiliary set of address-taken variables that are potentially referenced by the pointer it is storing to (its `MayDef` set), and (ii) it returns a new virtual register for each input pointer that is either directly-written or part of the `MayDef` set and replaces successive uses of those input pointers with the new SSA variable names. Listings 3.1 and 3.2 demonstrate the transformation of `store` into `dfi.store`. In the example pointer `%g` might point to variable `%a`. Therefore the `llvm.store` at line 3 in Listing 3.1 is transformed into a `dfi.store` that produces two results (`%g0` and `%a0`) representing the post-store `%g` and `%a` registers, with `%a` being in the `MayDef` set. As a consequence, e.g. the input argument of `load` in line 4 is updated to `%a0` to capture side effects induced by potential aliasing. In the absence of pointer aliasing the `MayDef` set is empty and `dfi.store` simply produces a new SSA value for the pointer operand that replaces its subsequent uses.

To capture side effects of read-only memory operations like `load` or `getelementptr` on pointers operands, we introduce `dfi.may_use` as an additional instruction. `dfi.may_use` makes potential side effects explicit by aggregating them into a new virtual register which is used to replace the original operand of the read-only memory operation. In line 5 and 6 of Listing 3.2 the input of the `load` instruction is replaced with the return value of `dfi.may_use` which aggregates side effects on `%a` through `%g`.

```
1 // %g might point to %a
```

```
2 func @f(%p: !ptr<i32>, %g: !ptr<i32>, %a: !ptr<i32>) {
3     llvm.store 34, %g
4     %v1 = llvm.load %a
5     %v2 = llvm.load %g
6 }
```

Listing 3.1: Original IR

```
1 // %g might point to %a
2 func @f(%p: !ptr<i32>, %g: !ptr<i32>, %a: !ptr<i32>) {
3     %g0, %a0 = dfi.store 34, %g mayDef{%a}
4     %v1 = llvm.load %a0
5     %m = dfi.may_use(%g0, %a0)
6     %v2 = llvm.load %m
7 }
```

Listing 3.2: IR after conversion to use `dfi.store` and `dfi.may_use`

`dfi.call` operation

To support interprocedural analysis, it is essential to capture value-flows of the callee function at a given call site. For callees that have no side effects (e.g. pure functions), it suffices to propagate value-flows through function arguments and the return value. However, we also need to consider *output arguments* where results are carried through pointer type function parameters. In order to capture the value-flows propagated through output arguments, DFI replaces normal function call operations, `llvm.call`, with custom `dfi.call` operations.

Similar to `dfi.store`, for each pointer argument in the original `llvm.call`, we add a pointer-type result in the result list of `dfi.call`. Listing 3.4 and 3.3 show the result of replacing `llvm.call` with `dfi.call`. [2]

```
1 llvm.func @f(%v: i32, %p: !ptr<i32>) -> i64 {
2   %r = llvm.call @g(%v, %p)
3   %0 = llvm.load %p : !ptr<i32>
4   llvm.return %r : i64
```

[2]For better readability, we will omit the type notation of all `dfi.call` occurrences in rest of the paper.

```
5 }
```

Listing 3.3: IR before conversion to `dfi.call`.

```
1 llvm.func @f(%v: i32, %p: !ptr<i32>) -> i64 {
2   %r, %p0 = dfi.call @g(%v, %p)
3   %0 = llvm.load %p0 : !ptr<i32>
4   llvm.return %r : i64
5 }
```

Listing 3.4: IR after conversion to `dfi.call`. The newly added function call result `%p0` on line 2 captures side effects induced by the callee `g` w.r.t pointer argument `%p`.

3.3.2 Structure and Workflow

Figure 3.2 shows the overall structure in DFI which comprises of two primary components: main analysis engine and client analysis. The main analysis engine performs whole-program analysis on the preprocessed MLIR code and passes results to the client analysis for downstream processing.

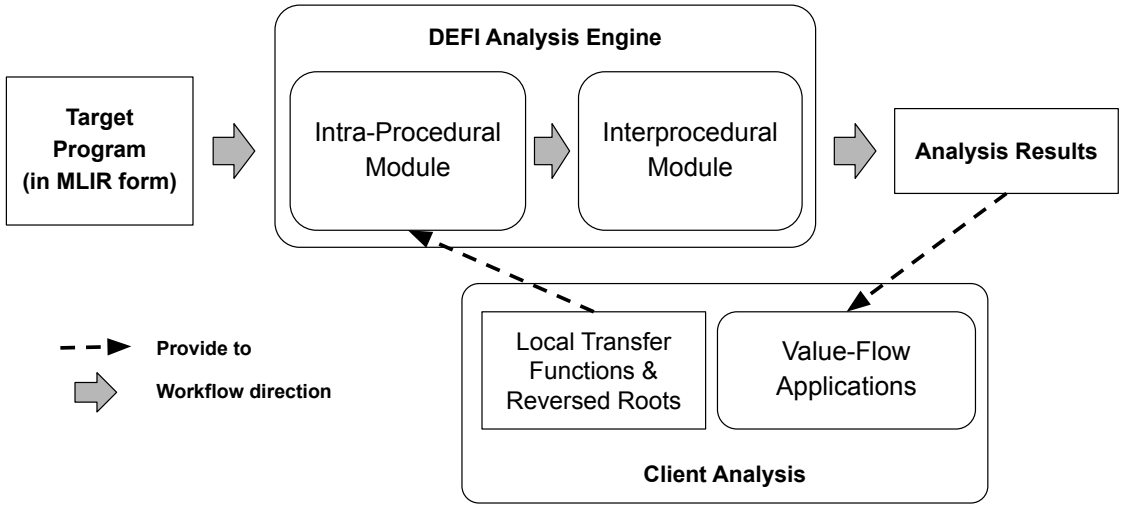


Figure 3.2: Structure of the analysis engine in DFI

First, as further detailed in Section 3.3.3, intraprocedural DFT intervals are computed by the analysis engine based on value-flow mapping information provided by the client analysis.

Next, as described in Section 3.3.4, the intraprocedural value-flows will be propagated to their callers in order to support interprocedural queries. Finally, the client analysis makes *analysis queries* to retrieve **flow-** and **context-sensitive** value-flow information from the analysis engine to solve domain-specific problems.

3.3.3 Intra-procedural Analysis

This Section discusses the algorithm of tracking intraprocedural value-flows. As mentioned in Section 3.2.2, value-flow analysis can be boiled down to a graph reachability problem, where each vertex is a dataflow fact on a certain program point and value-flow queries are answered by determining the reachability of two vertices.

Similar to IFDS, DFI also approaches the intraprocedural part of this problem by employing a local transfer function for each statement and a graph reachability solving technique. However, instead of the tabulation-based strategy used by the original IFDS technique, DFI adopts a novel graph reachability solving technique based on depth-first spanning tree intervals introduced in Section 3.2.3. The idea is that for a given value-flow analysis problem, if every relevant operation in the program is annotated with DFT intervals, we are able to determine the reachability between two arbitrary operations by comparing their intervals in nearly constant time.

To demonstrate this concept, Figure 3.3 shows a MLIR function in which the values are annotated with DFT intervals. For instance, value $\%p$ and $\%1$ have $\langle 0, 5\rangle$ and $\langle 1, 4\rangle$ for their intervals, respectively. These intervals are annotated on the spanning trees derived from the **SSA def-use graph**, consisting of edges that go from a single SSA value definition to its uses.

[3]To simplify the code, in rest of the paper we "inline" the constants that are normally represented by dedicated `llvm.constant` operations. Also, we use `!ptr` in replacement of `!llvm.ptr` for pointer types.

```
                                ⟨6, 9⟩      ⟨0, 5⟩
    llvm.func @foo(%a: i32, %p: !ptr<i32>)->i32 {
      %t = llvm.alloca 1 x i32
⟨7, 8⟩ %0 = dfi.store %a, %t : !ptr<i32>
⟨1, 4⟩ %1 = llvm.load %p : !ptr<i32>
      %2 = dfi.store 3, %p : !ptr<i32>
⟨2, 3⟩ %3 = llvm.add 2, %1 : i32
      llvm.return %3 : i32
    }
```

Figure 3.3: DFT intervals in a MLIR function[3]

Assume a *taint analysis* is performed on Figure 3.3 and $\%p$ is a tainted value. To determine if the return value %3 is tainted it is sufficient to check if its interval $\langle 2, 3\rangle$ is subsumed by that of $\%p$ $\langle 0, 5\rangle$. In this case, since $\langle 0, 5\rangle \supseteq \langle 2, 3\rangle$ per our definition in Section 3.2.3, the return value is tainted. In fact, both %1 and %3 are tainted, as their intervals are both inside the subtree of $\langle 0, 5\rangle$. On the other hand, %0, whose interval is $\langle 7, 8\rangle$, is not tainted because it resides in a different DFT subtree rooted at $\langle 6, 9\rangle$.

To calculate the trees and associated intervals shown in Figure 3.3 the client analysis customizes the traversal mechanism introduced in Section 3.2.3. First, for each function, the client analysis chooses a set of root vertices to start the traversal. Note that the traversal timestamp in each function always starts from zero. Second, for each vertex in the function, a local transfer function is provided by client analysis to dictate the out-going vertices to visit next. In the context of taint analysis on Figure 3.3, the client analysis picks $\%a$ and $\%p$ as the traversal roots. For the local transfer function, the rules for each kind of operation are shown in Figure 3.4. Take the rule for `dfi.store` as an example, in which the result pointer $\%q$ is tainted only if the stored value $\%v$ is tainted. It effectively stops the DFT traversal to go from $\%p$ to $\%q$, but allows the path from $\%v$ to $\%q$.

The DFT interval scheme described in the previous example only details the computation of

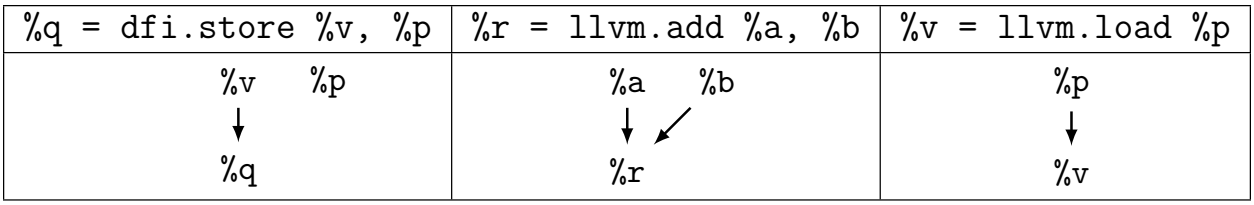


Figure 3.4: Local taint analysis transfer functions on Figure 3.3

reachability based on (spanning) *tree* edges. In the following Section we will discuss a more general reachability problem w.r.t both *tree and non-tree* edges.

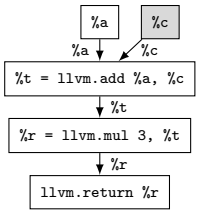
Non-Tree Edges in SSA Def-Use Graph

Non-tree edges inhibit the application of the simple interval-based graph reachability algorithm on generic SSA def-use graphs. To generalize our solution over those graphs, we first discuss program constructs that result in non-tree edges.

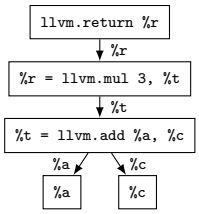
```
1 llvm.func @f(%a: i32, %b: i32, %c: i32) -> i32 {
2     %t = llvm.add %a, %c : i32
3     %r = llvm.mul 3, %t : i32
4     llvm.return %r : i32
5 }
```

Listing 3.5: A MLIR function

Figure 3.5a shows the SSA def-use graph for Listing 3.5. In Figure 3.5a, each vertex is labeled with an operation or value and edges are labeled with the value being used. Starting the creation of DFT intervals from function argument $\%a$ will produce three edges: $\%a$, $\%t$, and $\%r$. However, when we proceed to visit rest of the vertices, the edge between $\%c$ and *llvm.add* becomes a non-tree edge, since the latter operation has been visited before. We observe that, with this scheme, a non-tree edge appears if there is more than one operand in an operation. The additional operands create multiple parent vertices for the operation resulting in non-tree edges. Therefore, the number of non-tree edges is proportional to the number of operands. Unfortunately, the majority of nodes in standard LLVM IR have more



(a) SSA def-use graph



(b) Reversed SSA def-use graph

Figure 3.5: SSA def-use graphs for Listing 3.5

than one operand.

Benchmark name	no use	1 use	2 uses	3 uses	4+ uses
libcrypto (OpenSSL)	31.89%	59.65%	4.00%	1.53%	2.93%
libssl (OpenSSL)	33.44%	59.97%	2.30%	1.55%	2.74%
SQLite 3	30.70%	52.26%	9.53%	3.09%	4.42%
FFmpeg	20.67%	59.98%	11.53%	3.59%	4.23%
Lighttpd	30.50%	63.74%	2.08%	1.05%	2.63%
Servo	37.19%	50.58%	7.43%	1.69%	3.11%

Table 3.1: Percentage of SSA values with the respective number of uses

Reversed DFT Traversal

To effectively reduce the number of non-tree edges in a SSA def-use graph, DFI adopts a novel solution: traversing the graph in the *reverse direction* when building DFT intervals. In Figure 3.5a the traversal direction goes from a SSA definition to its uses. If we reverse this direction and go from a SSA use to its value definition, as demonstrated in Figure 3.5b, the number of non-tree edges no longer depends on the fixed number of operands. Instead, for a given value, the number of non-tree edges is proportional to the number of its SSA *uses*.

This led us to an important new realization: in most of the real-world codebases, the majority of the SSA values have a *single* or even *no SSA use*. Table 3.1 shows the percentage of SSA values against different number of SSA uses in 6 real-world codebases. The statistics show

that **80%** to **90%** of the SSA values have a single or no use in all benchmarks. In other words, if we traverse the SSA def-use graphs in the opposite direction, the resulting number of non-tree edges is far smaller than the number of tree edges.

Reversed DFT Root Reversed DFT traversal starts from a set of vertices called reversed roots. They are the out-neighbors of pseudo vertex δ introduced in Section 3.2.3. In DFI, the client analysis can pick its own reversed roots within a function body to begin the traversal. By selecting all possible value-flow end points as reversed roots, the client analysis ensures that all possible value flows through a function are expressed in the resulting graph, which is critical to the soundness of the analysis. Some of the most common end points include instructions producing no result (*e.g.* return instructions) and values with no SSA users. It is worth noting that DFI is able to compute all possible value flow paths at once because our algorithm is both time and space efficient, which we will show in Section 3.4.

Augmented DFT-Interval Graph Reachability

With reduced number of non-tree edges, DFI augments the DFT-interval-based algorithm to solve graph reachability in reversed SSA def-use graphs. In short, this method *duplicates* the interval upon encountering a non-tree edge, such that we can use a similar subsuming relationship between two intervals to determine their reachability.

To support our method, we introduce a new data structure: interval set. An interval set Π is a collection of intervals

$$\{\langle s_1, e_1\rangle, \langle s_2, e_2\rangle, ..., \langle s_n, e_n\rangle\}$$

in which every element separates themselves with each other by least one timestamp. *i.e.*

$$\forall \langle s_i, e_i\rangle, \langle s_j, e_j\rangle \in \Pi, i \neq j \implies e_i < s_j - 1 \vee e_j < s_i - 1$$

An interval set Π_i is said to subsume another set Π_j, namely $\Pi_i \supseteq \Pi_j$, if any of the interval in Π_i can subsume another interval in Π_j. *i.e.*

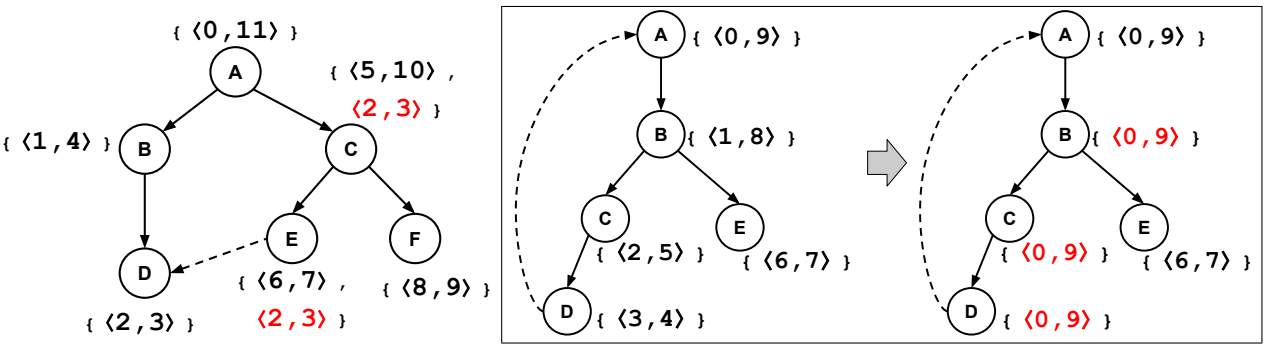
$$\exists \langle s_k, e_k \rangle \in \Pi_i, \langle s_l, e_l \rangle \in \Pi_j \ s.t. \ \langle s_k, e_k \rangle \supseteq \langle s_l, e_l \rangle$$

Two interval sets Π_i and Π_j can be merged by operator $\cup$, denoted as $\Pi_i \cup \Pi_j$. Let Π_k be the merged interval set from Π_i and Π_j, it can subsume both Π_i and Π_j *i.e.* $\Pi_k \supseteq \Pi_i \wedge \Pi_k \supseteq \Pi_j$. In our augmented DFT-interval-based reachability algorithm, each SSA def-use graph vertex v is associated with an interval set Π_v (rather than a single interval). Vertex v_i can reach v_j if and only if Π_{v_j} subsumes Π_{v_i}. *i.e.*

$$v_i \rightsquigarrow v_j \iff \Pi_{v_j} \supseteq \Pi_{v_i} \wedge \Pi_{v_i} \neq \emptyset \wedge \Pi_{v_j} \neq \emptyset$$

Note that since we build DFT intervals in *reversed* direction, a vertex can reach another vertex if the interval set of the destination subsumes that of the source. To build interval sets for each vertex in the graph, the interval set of each vertex is first initialized with a single interval created from a normal DFT-interval building process (see Section 3.2.3). Next, non-tree edges are incorporated.

Given a cross edge $v_s \rightarrow v_d$, Π_{v_d} is merged into Π_{v_s} and the interval sets of all of its ancestors. Figure 3.6a shows an example of handling cross edge $E \rightarrow D$. The interval set for destination vertex D, $\{\langle 2, 3 \rangle\}$, is merged into the interval sets of E, as well as its ancestors C and A. This allows us to account for graph reachability of two vertices that passes through $E \rightarrow D$. For example, C can reach D because $\Pi_C \supseteq \Pi_D$. If $v_s \rightarrow v_d$ is a back edge, Π_{v_d} will be merged into the interval sets of all vertices in the corresponding Strongly-Connected Component (SCC). The intuition behind this is that every vertex in such SCC belong to a subtree rooted at v_d, per the definition of back edge mentioned in Section 3.2.3. In other words, Π_{v_d} subsumes the interval sets of all of those vertices. Thus, our merging scheme here is able to reflect the



(a) Cross Edge (b) Back Edge

Figure 3.6: Non-tree edge handling

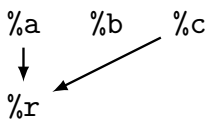
mutual connectivity of vertices in a SCC, including the back edge. Figure 3.6b shows an example of handling back edge $D \rightarrow A$. The interval set for destination vertex A, $\{\langle 0, 9\rangle\}$, is merged into the interval sets of every other vertex in the same SCC. Namely, vertices B, C, and D.

Meet Operator The meet operator specifies how value flows are combined from different program paths *e.g.* at a control flow merge point. In our algorithm, we use interval set merge $\cup$ as our meet operator. Per our previous definition of $\cup$, the merged interval set is always a safe approximation for the incoming interval sets and thus ensures soundness.

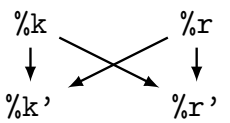
3.3.4 Inter-procedural Analysis

This Section discusses how to apply our interval-based reachability algorithm across interprocedural constructions *e.g.* call sites. DFI uses a summary-based interprocedural value-flow algorithm. We summarize the value-flows of each function in a *function value-flow summary* which is propagated to all of its call sites. Since the newly introduced callee summary alters value-flows inside the caller function, this process is repeated until a fixed point is reached.

A function value-flow summary comprises of a set of reachable relationships between function

Value	Interval Set	Summarized Value-Flow
%r	$\{\langle 0,7\rangle\}$	
%t	$\{\langle 1,6\rangle\}$	
%a	$\{\langle 2,3\rangle\}$	
%c	$\{\langle 4,5\rangle\}$	

(a) Listing 3.5

Value	Interval Set	Summarized Value-Flow (On Call Site)
%t1	$\{\langle 0,7\rangle\}$	
%t0	$\{\langle 1,4\rangle\}$	
%k	$\{\langle 2,3\rangle\}$	
%r	$\{\langle 5,6\rangle\}$	

(b) Listing 3.6, reversed root %t1

Figure 3.7: Function value-flow summaries in two different synthetic analyses

arguments and (outgoing) results. The results of a function are returned values or output (*i.e.* pointer) arguments. The summary describes the mapping between arguments and results which is expressed through argument and result indices. The result index is equal to the index of its counterpart in the result list of the `dfi.call` operation (see Section 3.3.1). In every `dfi.call`, the original returned value (if there is any) has index 0, followed by pointer argument type results. We denote $I(p)$ as the argument index of argument p. In addition, $O(p)$ is defined as the result index of p if p is part of PT_f, a subset of function arguments for f containing all pointer arguments. The value-flow summary for function f is denoted as $S_f = S_f^R \cup S_f^P$. The sets S_f^R and S_f^P represent mappings from function arguments to return value and to output arguments respectively. Let R_f and P_f be the set of return values and arguments in f, respectively. S_f^R is defined as

$$\{I(p) \rightsquigarrow 0 \mid p \rightsquigarrow r, \forall p \in P_f, \forall r \in R_f\}$$

Figure 3.7a shows the value-flow summary of `f` based on interval sets derived from a synthetic value-flow analysis. Since $\Pi_r \supseteq \Pi_a$ and $\Pi_r \supseteq \Pi_c$, the value of $S_f^R == S_f$ is $\{0 \rightsquigarrow 0, 2 \rightsquigarrow 0\}$

```
1 llvm.func @g(%k: !ptr<i32>, %r: !ptr<i32>) {
2     %t0 = llvm.load %k : !ptr<i32>
3     %t1 = dfi.store %t0, %r : !ptr<i32>
4 }
```

Listing 3.6: A MLIR function with two pointer arguments

S_f^P captures the potential value-flows applied on pointer arguments in function f. Specifically, it contains mappings between pointer arguments and corresponding output results on the `dfi.call` sites. Since the result of a pointer argument can be difficult to locate in the callee, we use the reachability between pointer arguments and the reversed roots as a safe approximation[4]. For arbitrary two pointer arguments $p0$ and $p1$, S_f^P is initialized with $I(p0) \rightsquigarrow O(p0)$ and $I(p1) \rightsquigarrow O(p1)$. If $p0$ and $p1$ can both reach a reversed root τ, then we can add $I(p0) \rightsquigarrow O(p1)$ and $I(p1) \rightsquigarrow O(p0)$ into S_f^P. Formally speaking, let T_f be the set of reversed roots in f, S_f^P is defined as

$$\{I(p) \rightsquigarrow O(k), I(k) \rightsquigarrow O(p) \mid \exists \tau \in \mathrm{T}_f \ s.t.$$
$$p \rightsquigarrow \tau \wedge k \rightsquigarrow \tau, \forall p, k \in PT_f\} \cup \{I(p) \rightsquigarrow O(p) \mid \forall p \in PT_f\}$$

Figure 3.7b shows an example value-flow summary for function `g` in Listing 3.6 based on another synthetic value-flow analysis. k' and r' correspond to the results of k and r at a call site, respectively. Since both k and r can reach the reversed root `%t1`, S_g^P in this case can be written as

$$\{I(k) \rightsquigarrow O(k), I(k) \rightsquigarrow O(r), I(r) \rightsquigarrow O(r), I(r) \rightsquigarrow O(k)\}$$

which is equal to $\{0 \rightsquigarrow 0, 0 \rightsquigarrow 1, 1 \rightsquigarrow 1, 1 \rightsquigarrow 0\}$

Value-Flow Summary Propagation

Next, we propagate the value-flow summary S_f to every call site of f. On a high level, S_f provides the missing local value-flow mappings at every call site of f. With this information in place, we are able to add new value-flows to the caller functions to improve precision. This process consists of two phases.

In the first phase, for each $v_s \rightsquigarrow v_d$ in S_f, we perform another round of reversed DFT

[4]As explained in Section 3.3.3 all value flow endpoints are selected as reversed roots.

f		g	
Value	Interval Set	Value	Interval Set
%v	$\{\langle 5,6\rangle\}$	%k	$\{\langle 3,4\rangle\}$
%p	$\{\langle 1,2\rangle\}$	%b	$\emptyset$
%t0	$\{\langle 0,3\rangle\}$	%s0	$\emptyset$
%t1	$\{\langle 4,7\rangle\}$	*%s1*	$\{\langle 0,1\rangle\}$
$S_f = \{0 \rightsquigarrow 1, 1 \rightsquigarrow 0, 1 \rightsquigarrow 1\}$		%s2	$\emptyset$
		%s3	$\{\langle 2,5\rangle\}$

(a) After intraprocedural analysis

f		g	
Value	Interval Set	Value	Interval Set
%v	$\{\langle 5,6\rangle\}$	%k	$\{\langle 3,4\rangle\}$
%p	$\{\langle 1,2\rangle\}$	%b	$\{\langle 8,9\rangle\}$
%t0	$\{\langle 0,3\rangle\}$	%s0	$\{\langle 3,4\rangle, \langle 6,7\rangle\}$
%t1	$\{\langle 4,7\rangle\}$	*%s1*	$\{\langle 0,1\rangle\}$
$S_f = \{0 \rightsquigarrow 1, 1 \rightsquigarrow 0, 1 \rightsquigarrow 1\}$		%s2	$\emptyset$
		%s3	$\{\langle 2,5\rangle\}$

(b) After value-flow summary propagation phase 1

f		g	
Value	Interval Set	Value	Interval Set
%v	$\{\langle 5,6\rangle\}$	%k	$\{\langle 3,4\rangle\}$
%p	$\{\langle 1,2\rangle\}$	%b	$\{\langle 8,9\rangle\}$
%t0	$\{\langle 0,3\rangle\}$	%s0	$\{\langle 3,4\rangle, \langle 6,7\rangle\}$
%t1	$\{\langle 4,7\rangle\}$	*%s1*	$\{\langle 0,1\rangle, \langle 8,9\rangle\}$
$S_f = \{0 \rightsquigarrow 1, 1 \rightsquigarrow 0, 1 \rightsquigarrow 1\}$		%s2	$\{\langle 3,4\rangle, \langle 6,7\rangle\}$
		%s3	$\{\langle 2,5\rangle\}$

(c) After value-flow summary propagation phase 2

Figure 3.8: Interval sets breakdown for Listing 3.7. Values with italics names are reversed roots.

traversal in the *caller* context. This time, instead of using reversed roots designated by the client analysis, we use the actual parameter values of v_s as revered roots. Upon finishing, the second phase starts by propagating the resulting interval sets of v_s from v_d in the *opposite* direction until reaching the original (caller) reversed roots.

```
1  llvm.func @f(%v:i32, %p:!ptr<i32>) -> i32 {
2      %t0 = llvm.load %p : !ptr<i32>
3      %t1 = dfi.store %v, %p : !ptr<i32>
4      llvm.return %t0 : i32
5  }
6  llvm.func @g(%k:i32, %b:!ptr<i32>) -> i32 {
7      %s0 = llvm.add %k, %k : i32
8      %s1,%s2 = dfi.call @f(%s0, %b)
9      %s3 = dfi.store %k, %s2 : !ptr<i32>
10     llvm.return %s1 : i32
11 }
```

Listing 3.7: Snippet for value-flow summary propagation.

Listing 3.7 along with Figure 3.8 demonstrates the interprocedural propagation of intervals. Applying the local transfer functions from Figure 3.4 results in (callee) value-flow summary $S_f = \{0 \rightsquigarrow 1, 1 \rightsquigarrow 0, 1 \rightsquigarrow 1\}$ (See Figure 3.8a). On line 8, phase 1 performs a reversed DFT traversal using `%s0` and `%b` (*i.e.* actual parameter to `%v` and `%p` in `f`) as reversed roots (see Figure 3.8b). In phase 2, the interval sets for `%s0` and `%b`, obtained in phase 1, are transitively merged into the interval sets of its parent vertices (*i.e.* `%s1` and `%s2`) in the augmented DFT (see Figure 3.8c).

Propagating callee value-flow summaries into the caller function might change its value-flow summary. DFI uses a worklist-based algorithm, shown in Algorithm 1, to recursively propagate the changed summaries to its callers until reaching a fixed point.

Algorithm 1 Interprocedural Worklist Algorithm

```
1: Worklist = all functions
2: while Worklist ≠ ∅ do
3:     f := Worklist.pop()
4:     S_f := GetVFSummary(f)
5:     if S_f has changed then
6:         for all c ∈ callers(f) do
7:             PropagateSummary(S_f, c)
8:             if c ∉ Worklist then
9:                 Worklist.push(c)
10:            end if
11:        end for
12:    end if
13: end while
```

In Algorithm 1, $GetVFSummary(f)$ returns S_f for function f; $PropagateSummary(S_f, c)$ performs the two-phase propagation introduced earlier with callee value-flow summary S_f in caller function c.

Reachable Functions Summary

We are now able to calculate interprocedural value-flows with flow and context-sensitivity of two values v_a and v_b in the *same* function by determining their reachability via interval sets *i.e.* $\Pi_{v_b} \supseteq \Pi_{v_a}$. In this part, we generalize this ability for v_a and v_b that reside in arbitrary functions.

In the previous Section, the value-flow summary of callee function was propagated to the caller, in order to add context-sensitive value-flows into the caller function. This process is now augmented to additionally propagate a *reachable function summary*. A reachable function summary $\Psi(\varepsilon)$ provides a mapping from an endpoint ε to a set of *transitively-reachable* endpoints. An endpoint ε_f^i represents the i-th argument of function f.

```
1 llvm.func @f(%a0: i32, %a1: i32) {
2     dfi.call @g(%a0)
```

```
3 }
4 llvm.func @g(%a0:i32) {
5     dfi.call @f(10, %a0)
6     dfi.call @k(23, 75, %a0)
7 }
8 llvm.func @k(%a0:i32, %a1:i32, %a2:i32)
```

Listing 3.8: MLIR functions with interprocedural calls

Take Listing 3.8 as an example, the reachable function summary for ε_f^0 (*i.e.* the first argument of function `f`) is

$$\Psi(\varepsilon_f^0) = \{\varepsilon_g^0, \varepsilon_f^1, \varepsilon_k^2\}$$

Because the first argument of `f` will be passed to the first argument of `g`, which further passes it to the second and third argument of `f` and `k`, respectively.

To calculate Ψ, we use the same infrastructure outlined in Algorithm 1. Basically, for a given function f with n arguments, $\Psi(\varepsilon_f^{0\ldots n-1})$ is repeatedly propagated to all of its callers and merged with their summaries until reaching a fixed point, namely, none of the reachable function summaries changed. With reachable function summaries, we are able to perform a two-stage process to answer the value-flow query $v_a \rightsquigarrow v_b$, where v_a and v_b are in different functions f and g, respectively. First, we calculate a set

$$\{\varepsilon_e^i \mid \forall e \in \mathit{callees}(f)\ \mathit{s.t.}\ v_a \rightsquigarrow \varepsilon_e^i\}$$

Then, we check if any of the $\Psi(\varepsilon_e^i)$ contains an endpoint of g, ε_g^j, for arbitrary argument index j. If not, that means function f cannot even reach function g. Finally, in the second stage, we can conclude that v_a can reach v_b only if endpoint ε_g^j can reach v_b. Namely, $\varepsilon_g^j \rightsquigarrow v_b \implies v_a \rightsquigarrow v_b$.

Subject	Version	# LOC	# LOC (LLVM IR)	# of Functions	Description
FFmpeg	4.2	1.2M	4.6M	17802	A/V encoder and decoder
OpenSSL	3.0.0	532K	1.1M	13490	Crypto and TLS
SQLite 3	3.36.0	166K	376K	1103	SQL database
Nginx	1.23.1	152K	502K	1415	Web server
Lighttpd	1.4.60	97K	216K	1258	Web server
du	9.1.113	816	74K	577	Disk utility
expr		779	58K	405	Command line utility
csplit		1063	53K	386	Text processor
tac		481	49K	334	Text file utility
ls		4K	42K	561	Filesystem utility

Table 3.2: Description of analysis subjects we used in scalability evaluations

3.4 Evaluation

In this Section, we discuss three research questions:

- **RQ1:** The performance of the graph construction. Specifically, we evaluate the scalability of DFI against large codebases (§ 3.4.1 and § 3.4.2).
- **RQ2:** The efficiency of answering graph reachability queries. Specifically we examine the size distribution of vertex interval sets which is strongly related to query time (§ 3.4.3).
- **RQ3:** The precision of DFI which we measure in terms of false positives / negatives for a given client analysis (§ 3.4.4).

For **RQ1** and **RQ2**, we select five open-source codebases of different domains and sizes as our primary analysis targets: Lighttpd, Nginx, SQLite 3, OpenSSL, and FFmpeg; as well as five small programs from Coreutils: ls, tac, csplit, expr, and du. Table 3.2 details the selected target programs, lines-of-code (LOC) are listed in textual as well as LLVM 14 IR code, since DFI operates on LLVM IR input. The targets are built using Link-Time Optimization (LTO) to generate a monolithic executable without dynamic dependencies. As mentioned in

Section 3.3.1 DFI operates on MLIR. However, due to its novelty MLIR still lacks a general pointer analysis implementation at the time of writing. Thus, all of the experiments in this section omit information from supplemental pointer analysis. All experiments are performed on a commodity desktop running Ubuntu 20.04 LTS with a 6-core, 12-thread Intel i7-8700K CPU and 32GB of RAM.

3.4.1 Scalability of Different Client Analyses

We measured the performance of DFI with two different client analyses: **uninitialized variable analysis** and **taint analysis**, both analyses are flow and context-sensitive.

The uninitialized variable analysis determines if an address-taken variable is initialized before use. To simplify the problem without losing generality, we only consider memory stores as valid initializations of a variable. The DFI client analysis selects values without users and instructions with no result as the reversed roots in each function and constructs the value flow graph accordingly. We then query the constructed graph to determine if heap memory allocated by `malloc` remains uninitialized before use by a `strlen` function.

The taint analysis determines if a tainted value can flow from one point of the program (*i.e.* the source) to a different program point (*i.e.* the sink). We consider the destination of a memory store tainted only if the stored value is tainted. Reversed root selection is identical to the previous analysis; we select return values of `__errno_location` as source and first operands of `strlen` as the sinks.

Table 3.3 lists the results of running uninitialized variable analysis and taint analysis against all targets listed in Table 3.2. The runtime measurement is split into three categories, *Total time (TT)*, *Analysis time (AT)*, and *Query time (QT)*. The total time is a normalized wall-clock time of the entire process excluding file parsing; the analysis time measures the time

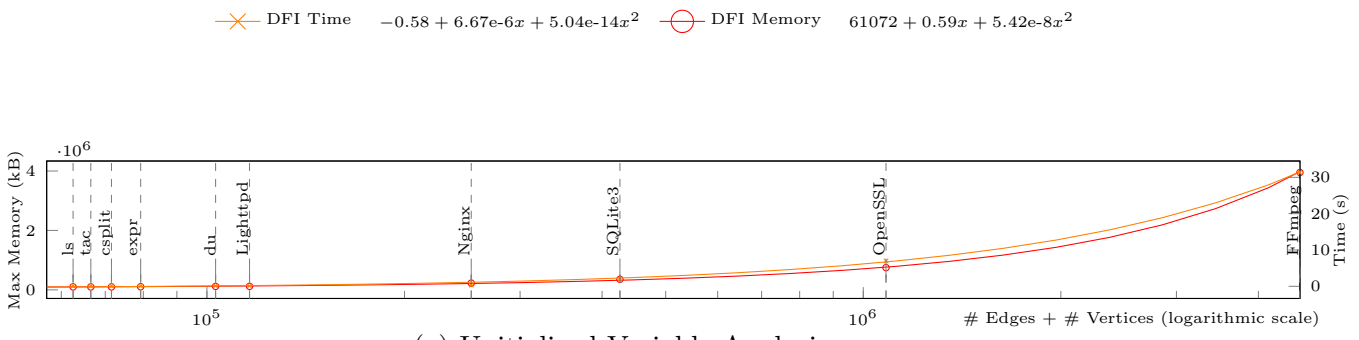
	Uninitialized Variable Analysis						Taint Analysis						PhASAR Taint Analysis	
	#V+#E	# Q	AT (s)	QT (ms)	TT (s)	Max RR	#V+#E	# Q	AT (s)	QT (ms)	TT (s)	Max RR	TT (s)	Max RR
FFmpeg	4627096	169260	5.41	1450.9	31.33	3.77 GB	6827728	166140	7.51	452.3	60.22	4.97 GB	900.48	15.73 GB
OpenSSL	1083094	682	0.61	0.6	6.98	729.88 MB	1475276	43648	2.10	421.5	13.29	1.04 GB	41.54	2.86 GB
SQLite3	425875	204	1.77	0.2	2.03	347.02 MB	476421	11016	2.27	8.5	2.76	404.77 MB	Out-Of-Stack	Out-Of-Stack
Nginx	252779	1380	0.17	4.6	0.32	223.12 MB	290343	11940	0.19	51.6	0.59	233.43 MB	3.30	601.37 MB
Lighttpd	116120	324	0.05	0.5	0.12	113.91 MB	163115	1638	0.12	6.1	0.25	140.39 MB	0.70	275.11 MB
du	103090	1248	0.07	1.7	0.13	113.71 MB	133390	1952	0.09	7.3	0.16	118.76 MB	0.41	205.66 MB
expr	79266	448	0.06	0.8	0.09	100.48 MB	103778	160	0.07	0.2	0.12	104.56 MB	0.31	166.41 MB
csplit	71550	336	0.05	0.5	0.08	95.84 MB	95216	348	0.06	0.4	0.10	98.79 MB	0.26	158.97 MB
tac	66550	336	0.05	0.6	0.08	95.73 MB	88626	288	0.06	0.4	0.11	98.88 MB	0.25	153.07 MB
ls	62538	630	0.04	1.0	0.09	96.11 MB	76408	2205	0.04	5.1	0.11	97.98 MB	0.25	144.52 MB

Table 3.3: Performance of different client analyses in DFI and PhASAR. (#V+#E: Number of visited vertices and edges; # Q: Number of queries; AT: Analysis time; QT: Query time; TT: total time; Max RR: Max resident memory)

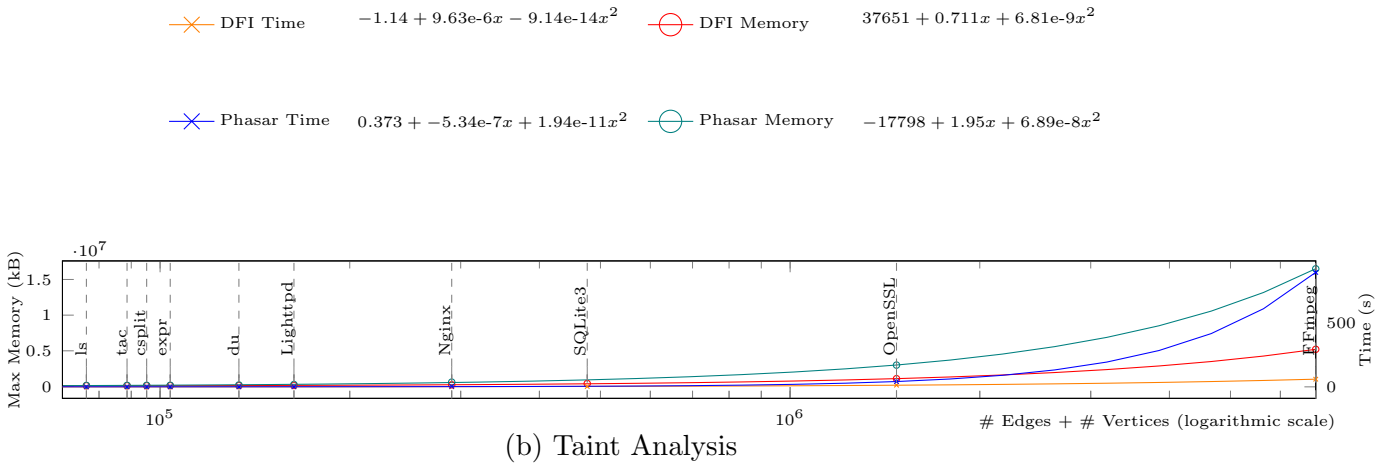
spent in the main analysis; query time measures the time spent in answering analysis queries based on the graph constructed in the main analysis phase. The memory consumption measures the maximum amount of physical memory allocated during the evaluation of each target, which is roughly equal to resident memory. The number of visited edges and vertices is listed in the *#V+#E* column, respectively. Our evaluation shows that taint analysis visited roughly $10\% \sim 56\%$ more vertices and edges compared to uninitialized variable analysis on every target. The number of analysis queries, as shown in the *# Queries* column, ranges from a few hundreds to hundreds of thousands for both analyses. All experiments finished within **2 minutes** using no more than **5 GB** of physical memory.

Scalability of the Analysis

To evaluate the scalability of DFI on our client analyses, we measure the increase in total time and memory consumption against increases of the target code size expressed as the sum of the number of edges and vertices in DFI's program graph. The results of the scalability evaluation w.r.t different client analyses are shown in Figure 3.9, along with their polynomial regression lines. Figure 3.9a shows the runtime (right y-axis) and memory consumption (left y-axis) of DFI-based uninitialized variable analysis against the total number of visited vertices and edges (in logarithmic scale) across all targets. The same experiment is repeated for taint analysis, shown in Figure 3.9b, which we also include measurements for PhASAR which we further discuss in Section 3.4.2.



(a) Unitialized Variable Analysis



(b) Taint Analysis

Figure 3.9: Scalability of time and memory consumption

In both experiments, DFI's runtime and memory consumption are quadratically bounded with a very small leading coefficient indicating *almost linear* growth, as that the linear coefficient dominates the increase in memory and runtime over the number of visited vertices and edges for the evaluated range. A similar trend is also observed on the analysis time of all targets except SQLite 3. We found that, in processing the SQLite 3 codebase, significantly more time was spent in the interprocedural worklist algorithm (Algorithm 1). Specifically, in the propagation of callee function summaries to call sites in caller functions. Further investigation reveals two important factors contributing to this problem. First, SQLite 3 has a much *denser* call graph, in which each callee function is called by 3x $\sim$ 4x more call sites on average, compared to call graphs in other targets. Second, Table 3.2 shows that functions in SQLite 3 contain more code on average, primarily because the codebase has fewer functions compared to other projects of comparable size. The fact that these two factors multiply together (*i.e.* higher number of propagations to many large-size caller functions) induces a higher performance overhead in Algorithm 1. However, we argue that the source code structure of SQLite 3, more specifically, their interprocedural function call structure, is relatively unusual.

Last but not the least, both analyses spend no more than negligible 0.01 *millisecond* per analysis query regardless of program size.

3.4.2 Comparison with PhASAR

PhASAR [53] is the state-of-the-art dataflow analysis framework for LLVM IR that can solve the same kind of problems as DFI. Namely, finite, distributive, subset value-flow problems. The last two columns of Table 3.3 list the performance of taint analysis as implemented by PhASAR. We use the same configuration as for DFI's taint analysis detailed in 3.4.1. To enable a fair comparison, we adapted PhASAR's taint analysis transfer

function to match DFI's. More specifically, the change consists of treating the result of `getelementptr` as tainted if any of its operands is tainted. Originally, PhASAR taints the result of `getelementptr` only if the base pointer is tainted. Note that even without this change, PhASAR shows nearly the same scalability behavior with the results in Section 3.4.2. As we will mention in Section 3.4.4, the ground truth in precision analysis uses the equivalent set of value-flow transfer rules as well. In line with the previous experiments, we also disable pointer analysis and exclude time spent on input parsing and preprocessing to evaluate PhASAR's performance.

As depicted in Table 3.3 DFI's taint analysis runs faster and takes less memory than PhASAR in nearly every benchmark. In addition, we found that PhASAR generally requires more stack space due to its deep call stack during dataflow propagation. Thus, we increase the maximal stack size from 8 MB to *512 MB* for this specific experiment. Nevertheless, PhASAR still runs out of stack when analyzing SQLite3. On average, DFI runs ∼**14x** faster and takes ∼**3x** less memory than PhASAR. If we focus on the five larger benchmarks, DFI outperforms PhASAR by ∼15x on performance and ∼3.6x on peak memory consumption. The runtime and memory consumption trends of DFI and PhASAR are also depicted in Figure 3.9b. PhASAR's memory and runtime requirements increase significantly faster than DFI's as expressed by the leading coefficient of the fitted polynomials which are ~ 10 and $\sim 10^3$ times larger for memory and runtime respectively.

3.4.3 Size Distribution of Interval Set

Section 3.3.3 introduces the concept of an interval set Π_v for a vertex v and its subsuming operator $\supseteq$. The properties of an interval set play an important role in efficiently determining the reachability of two vertices. Specifically, given two interval sets Π_p and Π_q with at most N intervals each, the time complexity for evaluating $\Pi_p \supseteq \Pi_q$ is $O(N^2)$.

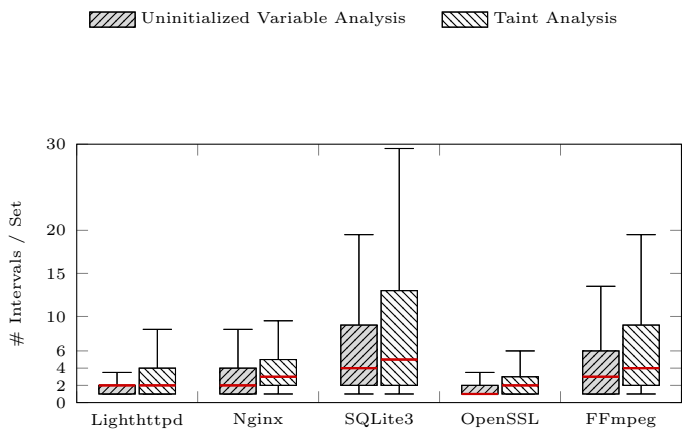


Figure 3.10: Size distribution of interval sets

Figure 3.10 shows the size distribution of interval sets across the five larger targets in both client analyses. In uninitilized variable analysis, majority of the interval sets have at most 20 intervals, while half of them have no more than 4 intervals. For taint analysis, half of the interval sets have at most 5 intervals and 75% of the interval sets have less than 15 intervals. These numbers show that the average size of an interval set is usually small. Thus, they offer additional evidence towards the effectiveness of our novel reversed DFT traversal scheme in reducing the number of non-tree edges.

3.4.4 Precision

In this section, we evaluate DFI's precision by evaluating taint analysis results on a selected set of 33 Coreutils programs and measure the number of false positives and negatives against ground truth. We also conduct the same experiment on PhASAR to compare with our results. The taint analysis we used in this experiment designates the `argc` argument of `main` function (representing the number of command line arguments) as the source and uses binary compare instructions in the target binary as sinks.

Retrieving Ground Truth

We obtain the ground truth for this experiment by a unique dynamic tracing technique based on sanitizers and symbolic execution. A center part of this technique is carried by DataFlowSanitizer [15] (DFSan for short) in Clang/LLVM. DFSan provides a instrumentation-based solution to dynamically track data flows. To that end, DFSan provides an interface to attach labels to target variables which are propagated along the program's dataflow paths and can be evaluated at arbitrary program points. For our experiments, we attach a label on the source variable (`argc`) at the beginning of the program; A sink (compare instruction) is considered tainted if any of its instruction operands contains the initial source label. To streamline the evaluation, we instrument compare/sink instructions with SanitizerCoverage [55] (Sancov for short), also provided by Clang/LLVM, to insert wrapper functions that check the labels on instruction operands and report taint status along with a unique tag assigned to the specific compare instruction.

As we generate our ground truth via dynamic tracing, we rely on high coverage to ensure the quality of our measurements. We use KLEE [18], a state-of-the-art symbolic execution framework, to generate a set of high coverage inputs and use them to drive the ground truth collecting runs. We run KLEE on each Coreutils tool for 2 hours, resulting in an average coverage of over 90%. The coverage for each tool is shown in the right-most column of Table 3.4. For the following evaluations we only consider sinks that are covered by our input corpus.

Comparison with PhASAR

Taint analysis precision for both DFI and PhASAR is shown in Table 3.4, presented as the number of false negatives, false positives, and true positives against ground truth. Contrary to the experiments in Section 3.4.1 and Section 3.4.2, pointer analysis is **enabled** for

PhASAR in this evaluation. Among these benchmarks, DFI has a lower precision of 66%, compared to PhASAR's 98% precision. However, PhASAR only has 39% recall, which is (much) lower than DFI's 90% recall and inferior to the average. Overall, the F1 score for DFI is **0.76**, which is higher than PhASAR's F1 score of **0.55**.

We found that majority of DFI's false negatives are due to limited support of global variables and indirect function calls, as well as the lack of supplemental pointer analysis information (see Section 3.5.2). We believe false negatives can be eliminated once these limitations are addressed. Most of the false positives are induced by foreign function calls whose value flow models in DFI are yet to be refined. For instance, both `memcpy` and `memmove` require better transfer functions to propagate tainted values. Another example is `malloc`: in DFI we consider the allocated heap memory to be tainted if its function argument (representing the size of the memory) is tainted. However, the ground truth always treats a heap memory freshly allocated by `malloc` as "clean" regardless of its function argument. We believe this issue can be easily solved by attaching custom transfer functions on foreign function calls.

3.5 Discussion

3.5.1 Soundness and Comparison with IFDS

DFI and the original IFDS [51] share many parts in their theoretical frameworks. Both of them adopt distributive transfer functions operate on the powersets of finite value-flow / dataflow facts and use the same meet operator. These properties imply the existence of fixed-point solutions, hence a safe approximation to the problem. DFI further restricts the value-flow facts to be SSA values, therefore it can not directly express the full range of dataflow problems expressible under IFDS, for example, conventional constant propagation. When it comes to interprocedural analysis, both works adopt a functional approach, namely,

Tool Name	DFI			PhASAR			**LCov**
	FN	**FP**	**TP**	**FN**	**FP**	**TP**	
basename	0	0	3	0	0	3	98.81%
comm	0	0	3	2	0	1	91.75%
date	0	0	3	4	0	3	92.53%
dircolors	1	0	2	3	0	0	85.47%
dirname	0	0	1	2	0	1	98.21%
echo	0	0	5	11	0	2	97.79%
expand	0	9	1	0	0	1	92.78%
false	0	1	0	0	1	0	97.06%
fmt	0	1	3	2	0	3	88.19%
hostid	0	0	1	0	0	1	97.22%
id	1	0	4	6	0	0	84.31%
kill	0	12	3	2	0	2	91.73%
link	0	0	3	0	0	3	95.12%
logname	0	0	1	0	0	1	94.74%
mkfifo	0	0	2	1	0	1	80.21%
mknod	0	0	6	6	0	2	86.03%
mktemp	0	0	2	3	0	0	94.81%
nice	0	0	3	4	0	3	90.91%
nl	0	0	2	0	0	2	92.05%
pathchk	0	0	2	1	0	2	82.05%
printenv	0	0	2	4	0	1	98.46%
readlink	0	0	3	2	0	1	98.78%
rm	0	0	3	3	0	2	95.33%
shuf	0	15	8	9	0	0	81.48%
sync	0	0	1	0	0	1	84.85%
touch	7	0	3	6	0	2	82.18%
tsort	0	0	2	1	0	1	95.43%
tty	0	0	1	0	0	1	98.25%
uname	0	0	1	0	0	1	90.83%
unexpand	0	1	1	0	0	1	92.09%
uniq	0	2	1	1	0	1	93.63%
unlink	0	0	2	0	0	2	97.44%
whoami	0	0	1	0	0	1	95.12%
Average	**0.26 ± 1.21**	**1.21 ± 3.51**	**2.32 ± 1.65**	**2.15 ± 2.76**	**0.03 ± 0.17**	**1.35 ± 0.95**	**91.95%**
Precision	**66%**			**98%**			
Recall	**90%**			**39%**			

Table 3.4: Precision of DFI and PhASAR on Coreutils programs (FN: false negatives; FP: false positives; TP: true positives)

leveraging callee function summaries.

The implementation of DFI is quite different from IFDS, though. DFI replaces the tabulation-based reachability algorithm with a simple interval lookup and avoids the accumulation of transitive edges which constitute a large part of the memory footprint of IFDS solvers [28, 9, 37]. Further, DFI processes only relevant program statements by utilizing sparse *SSA def-use chains*, whereas IFDS processes every program statement along *control flow paths* inducing memory and runtime overheads [28]. Since DFI has a completely different implementation, it's worth it to discuss the fixpoint convergence of DFI's intra- and interprocedural analysis algorithms.

Fixpoint convergence The intraprocedural reversed DFT traversal, introduced in Section 3.3.3, operates basically the same as normal DFS traversal on spanning tree edges and thus ensures its fixpoint convergence on those edges. While handling non-tree edges, we never remove any interval or interval subset during the propagation, therefore the process is monotonic with respect to the partial orders of the interval sets. Thus, fixpoint convergence is given. For the interprocedural case, we focus on Algorithm 1. The terminating condition for Algorithm 1 is dictated by changes to function summaries. These changes are caused by transitively propagating any callee function summary into the current function context. The propagation is driven by the same traversal algorithm that is also used for the intraprocedural case mentioned above. Thus, a fixpoint convergence w.r.t. function summaries can be inducted from the convergence of interval sets in callee functions, due to the fact that a function summary is derived from interval sets of the same function.

3.5.2 Limitations

Currently DFI is not fully capable of analyzing the following program constructions: variadic function calls, exception handling, and interprocedural value flows among global variables. We are actively working on extending DFI to add support for these features. In addition, DFI relies on auxiliary pointer analyses to calculate MayUse/MayDef pointer sets on memory operations. An over approximated pointer sets might indirectly create more non-tree edges in our reversed DFT and increase the average size of an interval set.

3.6 Summary

Value-flow analysis is an important component of many program optimizations. Previous researchers have made significant contributions and created important tools, but true scalability of these tools to large real-world codebases has so far proven elusive. We present a solution that is able to overcome these scalability bottlenecks. Key to our approach is a novel sparse graph representation of value flows that exhibits low tree widths. The resulting graph algorithms have much lower resource requirements and much better performance characteristics than previous approaches and provide almost linear scalability to truly large programs. Our prototype implementation is based on LLVM, is source-language agnostic, and will be open-sourced.

Chapter 4

Efficient Whole-Program Throughput Estimation

Differential throughput estimation, i.e., predicting the performance impact of software changes, is critical when developing applications that rely on accurate timing bounds, such as automotive, avionic, or industrial control systems. Often enough, developers have to create such applications without having continuous access to the target hardware, and hence they frequently need to rely on software-based instruction throughput estimation tools instead of being able to use accurate on-device measurements.

State-of-the-art estimation techniques broadly fall into two categories: On one side, there are dynamic approaches that emulate the execution of a program using cycle-accurate microarchitectural simulators. These achieve high precision, albeit at the cost of long turnaround times and convoluted setups, making them unsuitable for rapid development processes. On the other side are static approaches that predict cycle counts for a given stream of instructions outside of a concrete runtime environment. While such static approaches are fast, they don't scale well with the number of instructions under analysis and focus mostly on predictions

over single basic blocks, thereby requiring developers to pre-select critical instruction sequences manually. Furthermore, static approaches lack dynamic runtime information, which reduces prediction accuracy and precludes analysis of common programming constructs such as data-dependent control flows.

We present MCAD, a hybrid timing analysis framework that combines the advantages of dynamic and static approaches to provide accurate differential throughput prediction for complete programs. Instead of relying on heavyweight cycle-accurate emulation, MCAD collects instruction traces along with dynamic runtime information from QEMU and streams them to an LLVM based static throughput estimator. This results in an entirely new capability in which the performance impact of a software change can be estimated in minutes with high precision, reducing turnaround times by several orders of magnitude compared to existing approaches with similar accuracy. Our evaluation shows that MCAD scales to real-world applications such as FFmpeg and Clang with millions of instructions, achieving $< 3\%$ geo. mean error compared to ground truth timings from hardware-performance counters on x86 and ARM machines.

4.1 Introduction

Semantically equivalent modifications of a given piece of software can result in varying degrees of performance degradation due to resource contentions on the architectural and microarchitectural level. For systems that have tight timing restrictions, it is therefore critical to identify specific implementations that minimize negative performance impacts and maintain timing restrictions over the execution of the whole program. If the target system is not available to perform on-device measurements, developers instead need to rely on tools to estimate cycle counts for a given program or instruction sequence. To that end, several approaches have been developed that roughly fall into one of two categories: (i) emulating

the execution of concrete runtime instances of the program on simulated hardware (i.e., *dynamic approaches*) and (ii) estimating the cycle count of program instructions without concrete execution under an abstract runtime environment (i.e., *static approaches*).

Dynamic approaches achieve high precision using architectural simulators [17, 65, 13] which faithfully model the runtime behavior. They provide concrete and generally accurate estimates, however, they suffer from high runtime overhead and can require hours to days for analyzing a program. Furthermore, architectural simulators exhibit a high architecture dependence and are complicated to set up, often requiring dedicated expert knowledge and/or giving rise to compatibility issues with standard tools and default environments.

Static approaches [35, 43, 6, 2, 54, 39, 23, 40, 27] alleviate the performance and setup cost of dynamic hardware simulators. Traditionally, such approaches target worst-case execution time predictions over all possible execution paths [54, 39, 23, 40, 27]. More recent works [35, 43, 6, 2] focus on smaller execution sequences and also construct parametric models that generalize to multiple architectures. These models are either trained end-to-end using throughput data [43] or programmatically tuned for key parameters, such as port utilization, that are publicly available in the form of experimentally determined measurements [35, 6]. However, static approaches are fundamentally limited in practice due to their lack of concrete dynamic runtime information. Hence, traditional static approaches lack support for essential program constructs such as loops, data-dependent control flows, and memory accesses [44, 7]. In addition, the scope of many static throughput estimation tools is limited to throughput predictions of individual basic blocks [35, 43, 6], i.e., only a handful of instructions. Even for tools that can in theory process multiple basic blocks at once, predictions do not usually hold across control-flow transfers. For example, MCA [2] does not follow call or jump targets and instead simply falls through to the next instruction while adding a static cycle penalty[1].

[1] `https://github.com/llvm/llvm-project/blob/main/llvm/lib/MCA/InstrBuilder.cpp#L224`

In this paper, we present MCAD, a lightweight alternative that provides whole-program throughput prediction of binary software. MCAD follows a hybrid approach that supplements static throughput estimates with dynamic runtime information. To avoid the overhead of cycle accurate architectural simulation, MCAD uses an emulation-based approach to obtain execution traces using QEMU [12] and dynamically forwards them to the LLVM Machine Code Analyzer (MCA) [2] for instruction-level analysis. In addition, MCAD extends MCA to resolve several inherent limitations of static throughput estimation. First MCA's instruction analysis is redesigned to process instructions in a streaming fashion, which enables the analysis to scale to large real-world binaries. Second, concrete dynamic runtime information capturing control-flow and memory aliasing properties is incorporated into the analysis to obtain estimates that accurately predict instruction cycle counts across basic block boundaries. The main purpose of MCAD is to provide fast, yet accurate differential timing analyses: cycle counts for whole program execution traces which typically contain hundreds of thousands to millions of instructions can usually be produced within the order of minutes or seconds.

MCAD enables the assessment of timing effects induced by software changes as part of the development process by comparing the cycle counts before and after a change and identifying the least intrusive change with respect to execution time. We extensively test and evaluate MCAD with respect to scalability and accuracy on a number of different real-world applications such as FFmpeg and Clang to demonstrate that MCAD can model microarchitectural behaviors, such as instruction latencies in superscalar processors, accurately and with low cost. The geo. mean error in differential timing between MCAD and hardware performance counters in our experiments is smaller than 3% across several different microarchitectures and application software.

Summary of Contributions:

- We present MCAD: a new open-source[2] framework for throughput estimation yielding highly accurate differential timings, on par or better than the current state-of-the-art, while reducing turnaround time by several orders of magnitude.
- Our prototype implementation leverages QEMU as a fast instruction executor, utilizing MCA to model individual per-instruction execution cycles, rather than simulation-based approaches that faithfully model complex processor front-ends.
- We provide a detailed evaluation of MCAD with respect to accuracy and scalability for the popular x86 and ARMv8 instruction-set architectures using several different devices and hardware-performance counters to collect timing measurements for real-world traces as ground truth.

4.2 Background and Motivation

In this section we provide background on inherent limitations of static throughput estimation approaches and present a use case scenario to motivate the design goals of MCAD.

4.2.1 Static Throughput Estimation Challenges

Throughput estimation is an active area of research that aims to statically predict the performance upper bound of a program, usually measured by cycle counts or Instruction Per Cycle (IPC), of a *single* basic block. Current tools model microarchitectural details such as instruction latency and number of micro-ops of the target processor. However, there are two major issues with this approach: (i) it does not easily transfer across branch instructions or function call boundaries (ii) dynamic information such as execution context and memory

[2]Will be made available after de-anonymization. Some of our contributions in this work have been adopted and are currently in use as part of LLVM.

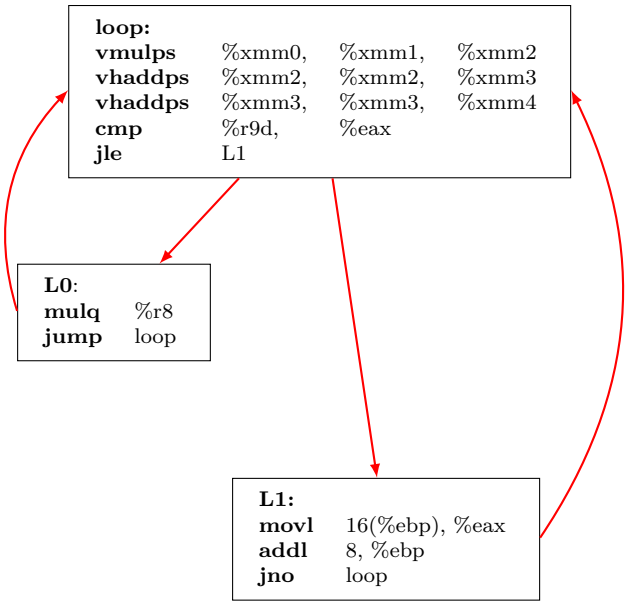


Figure 4.1: x86_64 assembly control flow.

aliasing is usually not taken into account.

Control Flow Transfers.

Figure 4.1 shows the control flow of an x86_64 assembly code snippet consisting of three basic blocks, `loop`, `L0`, and `L1`. Block `loop` calculates a vector dot product followed by a conditional branch into either block `L0` or `L1` based on a data-dependent comparison, with both blocks jumping back to `loop` at the end.

Using Intel Coffee Lake as an example target architecture, throughput predictions of the individual basic blocks of this program will not generalize across executions. The reason is that instruction-level throughput for this program actually depends on the ordering of executed basic blocks on that architecture. In particular, executions where `L0` follows `loop` (Listing 4.1) are roughly 5∼10 cycles slower, per iteration, than executions in which `L1` follows `loop` (Listing 4.2). This might seem counterintuitive as Listing 4.1 contains fewer instructions than Listing 4.2 and, more importantly, there is a memory read instruction (`movl`

`16(%ebp), %eax`) in the latter trace, which should be slower than scalar multiplication (`mulq %r8`). However, measurements on the target architecture reveal that there is a substantial slowdown in traces that follow the shorter Listing 4.1 due to resource contention between the two basic blocks.

```
1    vmulps   %xmm0, %xmm1, %xmm2
2    vhaddps  %xmm2, %xmm2, %xmm3
3    vhaddps  %xmm3, %xmm3, %xmm4
4    cmp      %r9d , %eax
5    jle      L1
6    mulq     %r8
7    jmp      loop
```

Listing 4.1: Trace of executing `L0` after `loop` in Figure 4.1

```
1    vmulps   %xmm0, %xmm1, %xmm2
2    vhaddps  %xmm2, %xmm2, %xmm3
3    vhaddps  %xmm3, %xmm3, %xmm4
4    cmp      %r9d , %eax
5    jle      L1
6    movl     16(%ebp) , %eax
7    addl     8 , %ebp
8    jno      loop
```

Listing 4.2: Trace of executing `L1` after `loop` in Figure 4.1

Specifically, the `vhaddps` instruction always requires execution port 5, which is also demanded by the `mulq` instruction [5]. This creates a dependency between those two instructions and forces the `mulq` instruction to stall until previous `vhaddps` instructions release the desired execution port. On the other hand, in Listing 4.2 `movl` and `addl` do not have conflicting resource requirements with the previous instruction. That means both instructions will be

dispatched into execution no later than the previous `vhaddps` instructions and execute in parallel, thus, resulting in higher instructions-per-cycle count than Listing 4.1.

As explained earlier, current static throughput prediction approaches will use static instruction ordering as a substitute for dynamic control flow, resulting in a low-accuracy prediction. Moreover, existing approaches face severe practical limitations with regards to scalability. In Section 4.3 we detail our design of MCAD which tackles both of these challenges.

```
1    vsetvli zero, a0, e8, m2, tu, mu
2    vadd.vv v12, v12, v12
3    vsetvl rd, rs1, rs2
4    vadd.vv v12, v12, v12
```

Listing 4.3: Example RISC-V assembly code

Execution Context.

Listing 4.3 shows a RISC-V assembly snippet comprised of vector instructions. In RISC-V, the `VSETVL` and `VSETVLI` instructions set the vector length multiplier (LMUL) which determines the number of elements that are processed by subsequent vector instructions. Therefore, the latency of each vector instruction differs depending on the current value of LMUL. For instance, line 1 in Listing 4.3 sets LMUL to 2 (due to the `m2` operand), which results in a cycle latency that reflects a LMUL of 2 for the `VADD` instruction in the next line; in line 3 LMUL is set to the value stored in register `rs2`, resulting in a potentially *different* latency that reflects the LMUL that was just set, for `VADD` at line 4. Due to the lack of dynamic runtime information, static approaches can not determine the concrete value of `rs2` and therefore can not accurately model the latency of the `VADD` instruction in line 4.

This example shows that while the instructions on line 2 and 4 are identical, their exact latencies are actually influenced by the environment values, namely LMUL, as well as dynamic

values stored in the registers. Current throughout prediction approaches fail to provide accurate estimations for these cases and resort to a conservative upper bound latency due to the lack of dynamic information. Some existing tools can circumvent this issue with manual annotations. For instance, LLVM MCA allows developers to instrument their programs with special comments that contain runtime information, which MCA uses to make more accurate queries into the scheduler model. However, while these instrument comments can improve analysis, handwriting them does not scale well with large number of instructions.

Memory Aliasing.

Real processors reorder instructions to optimize instruction throughput. To that end, they analyze memory dependencies between individual load and store instructions to determine a valid instruction scheduling that minimizes contention. Static throughput estimators trying to model this behaviour are limited due to the lack of concrete memory aliasing information. For instance, Listing 4.4 shows two x86_64 instructions that access memories indexed by base registers `%r13` and `%r14`. Without knowing the exact values in these base registers, it's hard to know if memory aliasing prevents the instructions from being reordered, which makes a big difference in terms of latency. Some tools like MCA either assumes that individual memory operations never access aliasing addresses or that all memory accesses alias; both cases resulting in lowered prediction accuracy.

```
1    addq   7, 8(%r13)
2    movq   %r9, 8(%r14)
```

Listing 4.4: x86_64 assembly code with memory accesses

4.2.2 Differential Throughput Estimation

Differential throughput estimation is meant to predict the performance impacts of applying certain changes (*e.g.* software patches) on the target programs. To motivate MCAD's differential throughout estimating capabilities, we detail how MCAD can be applied to find the optimal variant of a security patch to a binary with tight timing requirements post deployment. In our scenario, a buffer overflow vulnerability due to a missing bounds check has been identified. To fix the vulnerability the developer needs to patch the missing check into the binary. As detailed in Section 4.2.1 the location of the patch as well as the specific assembly instructions can have a high impact on overall performance due to resource contentions on the micro architectural level. Developers therefore typically iterate through several semantically equivalent versions of a patch in order to minimize the performance degradation. MCAD provides developers with the means to quickly iterate through several versions of the target binary to estimate the resulting performance impact and triage potential bottlenecks without requiring access to the actual hardware.

To perform the throughput estimation, a developer runs the original binary as well as several patch candidates with concrete inputs through the MCAD pipeline and compares estimated cycle counts between versions. If required, MCAD allows to restrict the analysis to specific regions of the program (see Section 4.3.1). In addition, MCAD provides a timeline view which details each instruction's state transitions through the instruction pipeline (see Section 4.4.3). This information helps developers to triage performance degradations and guides them towards execution paths that are less sensitive to changes.

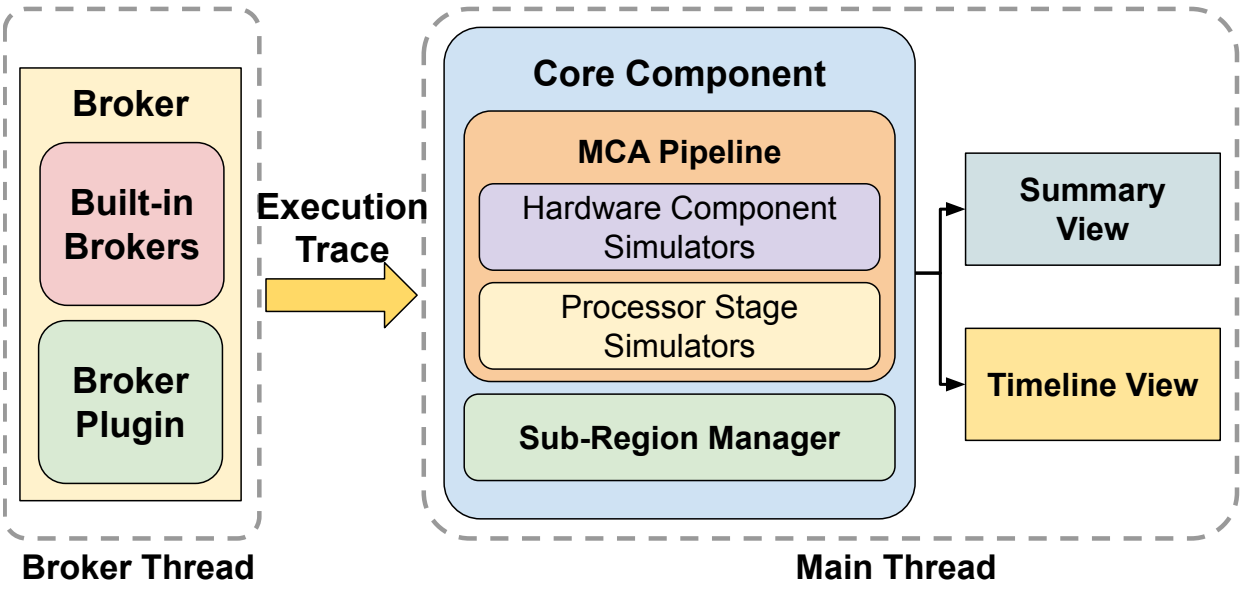


Figure 4.2: High-level structure of MCAD

4.3 Design

In this section we present our overall design of MCAD as depicted in Figure 4.2. As explained in the previous section, current throughput prediction approaches face severe challenges with respect to prediction across basic blocks, as well as scalability and turnaround times. The main goal of MCAD is to tackle all of these challenges to enable scalable and precise differential throughput analyses that can be used to actively drive development and steer engineers towards implementations with favorable runtime behavior.

4.3.1 Goals and Challenges

MCAD's design should tackle three main goals:

First, we would like to provide whole-program throughput estimates across thousands of basic blocks and potentially millions of instructions. At a high level MCAD uses a broker

component that provides execution traces in form of an instruction stream and a core component that analyzes instruction-level throughput of the respective trace on-the-fly. Results can then be processed by a viewer component for human-readable summarization and data reporting. In principle, the method by which execution traces are obtained and streamed to the core component is not tightly coupled to the method that is used to analyze the instruction stream. In early tests we compared several existing throughput analysis tools for use with our core component. However, we encountered several challenges with adopting any of them for our framework. As illustrated in the previous section, state of the art throughput prediction approaches do not generalize across control-flow transfers, and, within MCAD, streaming instructions that follow a control-flow transfer should seamlessly interface with the microarchitectural throughput prediction engine used for our analysis. As existing throughput prediction tools are designed for single basic block use, they also fail to scale up, in terms of both memory consumption and processing capabilities, when streaming input instructions on-the-fly from the broker component even for trivial programs. We also encountered numerous bugs when using the tools that seemed most fitting in this dynamic context, some of which we detail in Section 4.4.

Second, MCAD aims to support a development-driven workflow. This means, that developers are able to use MCAD to analyze the timing impact after modifying some part of the code, which might take the form of both a binary patch or a source-level change of the original program under our model. In addition to whole-program analyses, developers hence are able to choose to analyze only parts of the program. Selecting which parts of the program to analyze is done at varying levels of granularity to reduce noise in the resulting reports and speed up the analysis if so required. For example, in the scenario outlined by Section 4.2.2, the target program might contain components such as unmodified sequences of code, which are irrelevant to the throughput analysis, but whose run time might depend on unpredictable inputs (*e.g.* random number generators). In such cases, the developer can circumvent those components by excluding their traces from MCAD's analysis.

Third, MCAD aims to provide *timely* feedback for the throughput estimates using an approach that ideally also *generalizes* across architectures. While purely dynamic approaches are already capable of producing whole-program estimates today, the associated turnaround times and costs of setting up and running full-scale system simulation can be prohibitively expensive (i.e., on the order of hours or even days) [17, 65, 13]. Furthermore, existing dynamic throughput analysis tools are often tightly coupled to a specific architecture, which is why we opt for an emulation-based approach for producing execution traces inside our broker component using QEMU [12].

We will elaborate how MCAD tackles each of the respective challenges to achieve these goals throughout the rest of this section.

4.3.2 Scalable Throughput Prediction

As explained in Section 4.2.1 all existing throughput prediction engines are designed with single-basic-block use in mind. In our prototype we build on top of MCA [2], a performance analysis tool and library that was designed to estimate the basic block throughput in a static fashion. MCA employs a microarchitectural simulator to emulate an individual instruction's timeline inside an out-of-order processor. It taps into the LLVM compiler's scheduling database, a mature and production-tested data source whose contents are curated by hardware vendors.

However, we found that MCA has difficulties to scale up in our dynamic scenario. Like other throughput prediction engines, it does not support accurate analysis of instructions beyond a branch point or a function call out of the box. On top of that, we found that some of the design trade-offs inside MCA make it prone to high memory pressure while processing large number of instructions. Enabling online analysis within MCAD required several changes to the underlying analysis infrastructure, such as MCA's serialization, memory model, and

instruction lifecycle. For example, different memory operations were assumed to never access aliasing addresses. After introducing our changes, we found that dynamic memory traces can actually help the existing load-store unit inside MCA perform better.

In Section 4.4.2, we explain our modifications of MCA used in MCAD to tackle the aforementioned issues and make MCAD's core component scale up to real-world applications.

4.3.3 Development-Driven Workflow

MCAD enables a development-driven workflow by providing fast whole-program throughput estimates that can easily be compared between two versions of a program. Furthermore, for cases that only modify a small portion of the original binary, MCAD also provides an option to analyze only part of the binary. In this mode, developers can designate the desired area by either specifying the symbol of a function or providing explicit address ranges in the program. If an address range is provided, MCAD essentially yields the original basic-block granularity of the underlying analysis engine, while providing the flexibility of comparing the execution of multiple basic blocks at the same time.

4.3.4 Analysis Performance and Generalization Across Architectures

The broker component is responsible for supplying execution traces to the core component. This includes interoperating with the origin of execution traces and converting them into a unified low-level representation. This also means that the broker and core components need to work together to enable a timely and architecture-independent operation of MCAD. By default, execution traces are transmitted remotely from QEMU using a custom plugin. QEMU's emulation-based approach incurs around 30% runtime overhead [32], effectively en-

abling near-native execution speeds when using hardware virtualization extensions. QEMU also has extensive support for many major architectures,[3] meaning that MCAD's broker component is able to fulfill both of these requirements. It is noteworth to mention that MCAD also provides a facility for reading offline execution traces, which can be collected from executions on a physical device using any kind of tracing method available for that device. As mentioned earlier, MCAD's core component uses MCA. Since MCA uses LLVM's infrastructure, targeting different hardware architectures and processor models in the analysis engine requires little effort.[4] As a result, both the broker and the core component of MCAD generalize well across several architectures and provide top-of-the-line performance. Moreover, with QEMU and LLVM MCAD uses tools that many software developers will already be deeply familiar with.

4.3.5 Model Assumptions

MCAD is able to model out-of-order and superscalar execution commonly seen in modern processors. Simultaneous Multithreading (SMT), data or instruction cache simulation, and branch predictor are not in the current scope of this project. In addition, MCAD currently only supports analysis of single threaded executions.

4.4 Implementation

In this section we describe the workflow and implementation of MCAD in detail. Figure 4.3 depicts the interaction between the different components: first, the target binary program is executed by QEMU, which collects execution traces and sends them to our analysis engine in

[3]QEMU supports x86, MIPS, SPARC, ARM, PowerPC, RISC-V among many others, including individual processor models and specific microarchitectures

[4]LLVM supports x86, MIPS, SPARC, ARM, PowerPC, RISC-V among many others.

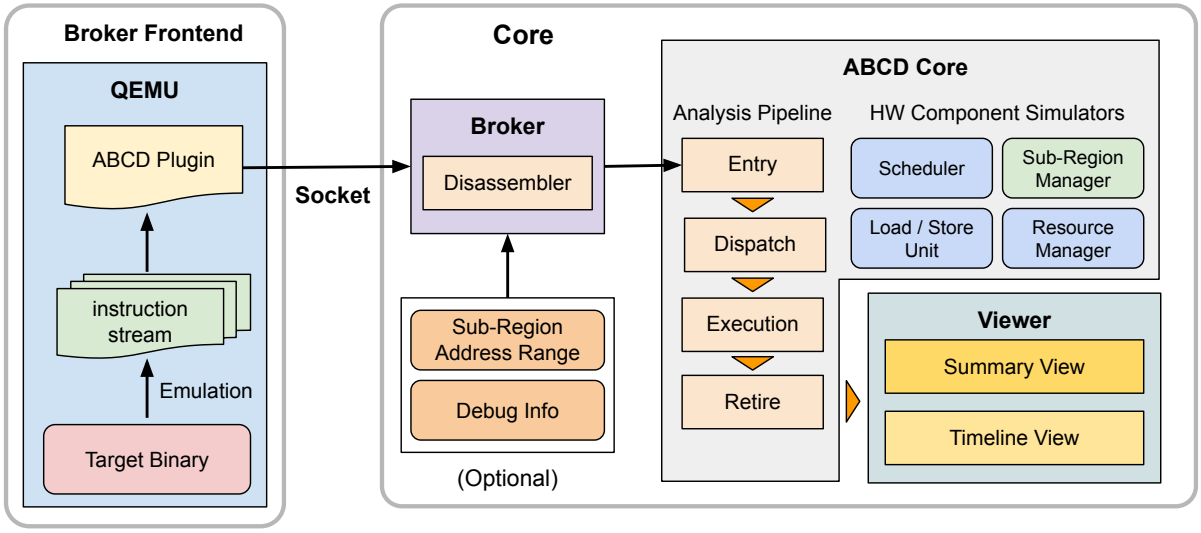


Figure 4.3: General workflow and interactions between the major components within MCAD.

real time. Inside our analysis engine the executed instructions are further processed by timing analyses built on top of MCA [2], which provides algorithms for microarchitectural simulation and instruction scheduling of modern processors. Finally, MCAD provides estimates for key timing and performance metrics like the prospective cycle counts, instruction-level throughput, and the ability to identify potential bottlenecks.

4.4.1 Instruction Broker

MCAD's broker implementation is a standalone process that produces `MCInst` [4] objects, an internal representation for machine code instructions used within LLVM, and forwards them in batches to the core component. The broker interface is designed to be extensible and allow integration of custom implementations and enable streaming of instruction sequences to the core component from a variety of different sources. So far, we integrated and tested two broker implementations: an assembly file broker that takes its input from an assembly file on disk and a QEMU broker that uses a QEMU plugin to communicate with the QEMU-broker process using TCP sockets to process the raw execution trace in real time before streaming them into the core component for analysis.

Broker implementations can choose to attach arbitrary metadata to the streamed instruction trace: for instance, by attaching load and store addresses and the size of memory operations we can enable more precise dependency detection between memory accesses in the analysis engine of the core component. In particular, MCAD's QEMU plugin collects raw instructions as executed by the emulator alongside additional information regarding memory operations, which is then used to improve analysis results with respect to instruction reordering. In this mode of operation MCAD's broker dynamically instruments memory read and write operations to gather target address and size of the data. The QEMU plugin will then send these data to the receiving core component that runs in parallel in a separate process. Inside the core component this metadata that is attached to memory operations is then inserted into a registry that is used by the core component for joint analysis.

Developing custom brokers is straightforward and only requires implementing a few callback functions before loading them as shared libraries during runtime. This allows users to rapidly switch between different workloads and environments depending on their needs. It is important to note that we do not make any assumptions about a broker's internal execution model – so long as the broker adheres to the streaming interface to supply the next batch of instructions.

4.4.2 Analysis Core

Our core component builds on top of state-of-the-art throughput prediction engines, which are designed as offline tools for static throughput estimation of small sequences of machine instructions (usually at the basic block granularity). Given a (short) sequence of assembly instructions, they provide throughput estimation results on the microarchitectural level either through end-to-end trained models for a given architecture or through simulation of the different stages inside a modern processor with varying levels of detail and manually tuned

key parameters per architecture. Unfortunately, all existing throughput prediction tools failed to scale up with our dynamic model of execution: as an example, using the standard video and audio encoder FFmpeg [58] executes around 20 million instructions on a Linux x86_64 machine while decoding a short MPEG-4 video with duration of 2 seconds. Within MCAD, this type of application would be considered a lightweight real-world workload. We found that none of the existing approaches were able to analyze anywhere near this kind of workload.

However, since MCA already has support for slightly larger pieces of code compared to all other related approaches through their *loop kernel* analysis, we implemented MCAD's default analysis engine on top of that. Our investigation into adopting MCA for our core component showed several failure cases while handling larger workloads. Internally, MCA models four distinct execution stages *Entry*, *Dispatch*, *Execute*, and *Retire*. Under MCAD's workflow the instruction stream provided by the broker enters from the Entry stage and is processed by each subsequent stage sequentially. MCA then assigns an internal data structure to each instruction to keep track of its scheduling status within the simulation pipeline. Originally, this pipeline reads all input instructions ahead of time before the start of the analysis. Because MCA assumes those instructions to come from a file. This property, while it aligns with the overall design goal of MCA to provide throughput estimates for only small sequences of assembler instructions, is not suited for whole program analysis. In some of our tests, MCA consistently drained all available physical memory on the machine running the analysis due to the allocation of this internal instruction representation.

Stream Processing To address the scalability issue we created a new incremental mode for the MCA simulation pipeline. In this mode, the simulation pipeline fetches input instructions incrementally. If there are no instructions available from the input source, the pipeline will save its current state and exit. Upon the arrival of new input instructions, the simula-

tion will be restored and proceed from its previous state. To reduce memory consumption, we implemented a new instruction recycling mechanism for the MCA simulation pipeline. This instruction recycler will reclaim and collect internal instruction data structures from retired instructions, instead of releasing their memory. These recycled data structures will then be reused to model new incoming instructions. Our experiments showed that with this recycling mechanism, MCAD's core analysis uses one third of memory on average than the unmodified MCA implementation.

MCA simulates different execution units which process individual instructions and determine results of the operation in question. These estimates include cycle counts, potential pipeline stalling, and predictions of possible instruction re-ordering. For instance, MCA's Load-Store Unit tracks the availability of memory operations and their (data) dependencies. This is crucial for simulating out-of-order scheduling in modern processors, which frequently reorder memory operations based on their dependencies. We significantly extended these existing capabilities by providing an *online* analysis workflow: by sequentially parsing the incoming instruction stream and processing each instruction according to the simulated pipeline, MCAD is able to present an estimate of how arbitrarily long instruction sequences might be scheduled within the processor.

Runtime Information As detailed in Section 4.2.1, due to unknown runtime state MCA's throughput predictions fail to model control flow transfers accurately and therefore struggle to analyze many essential programming constructs such as loops or interprocedural calls. MCAD addresses these issues by providing concrete runtime information that resolves ambiguities regarding block ordering, register state and memory aliasing.

Upon encountering a jump or call instruction MCA simply falls through to the next instruction as it lacks information about dynamic targets and fails to resolve data-dependent control flows. MCAD supplements this information based on the concrete execution traces

obtained from QEMU and forwarded to our analysis core.

MCA contains a component called Load Store Unit (LSUnit in Figure 4.3) which simulates load and store reordering that could happen in the hardware scheduler of the simulated processor model. This type of hardware optimization re-orders memory instructions to break dependencies when possible, which is largely determined by their memory aliasing properties at runtime. However, without precise memory access information, MCA can only make coarse-grained assumptions, for instance, all memory instructions are aliasing with each other, which are controlled by a command line parameter. MCAD leverages the memory traces collected from QEMU to improve this situation. We modified MCA's Load Store Unit such that aliasing properties are now dictated by fine-grained memory accessing traces as provided by our QEMU plugin. This enables our custom core component to simulate load and store reordering with higher accuracy by using dynamic information as it becomes available during execution.

4.4.3 Sub-Region Feature and Viewer Component

MCAD's viewer component displays throughput estimations with information like total cycle counts or potential pipeline stalls. An example of this can be seen in Listing 4.5. We also prototyped a view of the timing itinerary of individual instruction in a timeline view. For instance, Listing 4.6 shows the timeline of the execution trace in Listing 4.1. From this timeline we can easily spot the resource contention between `mulq` and `vhaddps` as mentioned in Section 4.2.1. This particular view is a fork from the timeline view that exists in MCA. However, the timeline view in MCA has limitations on the maximum number of analyzed instructions and cannot inspect the itinerary of a subset of an analyzed execution trace which we implemented as part of MCAD's subregion feature. For this reason, third-party visualization tools are also supported in this component. For instance, we prototyped a

new timeline view based on Chrome Developer Tools (DevTools) [1]. In our early test we already found this a lot easier to scroll and navigate through the thousands or even millions of instructions that are processed by MCAD, compared to the terminal-based LLVM-MCA timeline view. Since this view was designed to analyze large number of network requests it provides a solid basis to help our new timeline view scale up.

```
Instructions:      350
Total Cycles:      262
Total uOps:        600
Dispatch Width:    6
uOps Per Cycle:    2.29
IPC:               1.34
Block RThroughput: 5.0
```

Listing 4.5: Summary View

```
[1,0]     . D=eeeeE---------R ..   vmulps
[1,1]     .  D====eeeeeeE---R ..   vhaddps
[1,2]     .  D==========eeeeeeER   vhaddps
[1,3]     .   D==eE------------R   cmpl
[1,4]     .   D===eE-----------R   jle
[1,5]     .   D=====eeeeE------R   mulq
[1,6]     .   DeE--------------R   jmp
```

Listing 4.6: Timeline View

4.5 Evaluation

MCAD's main goal is to enable developers to quickly assess and iterate on the timing impact of program modifications, including *patches*, across control transfers (*e.g.* branches and function calls). In this section, we use binary programs of different release versions to evaluate performance and cycle-count accuracy against physical hardware traces to quantify how MCAD fares in comparison. More formally, given two different versions i and j of a program P, denoted as P_i and P_j, as well as a throughput predictor H that provides the number of execution cycles under a specific input for the respective program, we define the *differential throughput* $\Delta_H(P_i, P_j)$ describing the change in cycle counts between version i and version j of program P as predicted by H as follows:

$$\Delta_H(P_i, P_j) = \frac{H(P_j)}{H(P_i)}$$

Given two versions of a program P_i and P_j, as well as their inputs, we first use MCAD to predict their differential throughput, resulting in $\Delta_{MCAD}(P_i, P_j)$. Second, we similarly measure their relative difference in cycle counts from version i to version j using hardware-performance counters on physical devices, resulting in ground truth differential throughput $\Delta_G(P_i, P_j)$. Finally, we formally define the error of MCAD's prediction of differential throughput between version i and version j as:

$$E_{ij} = |\Delta_G(P_i, P_j) - \Delta_{MCAD}(P_i, P_j)|$$

4.5.1 Differential Throughput Prediction

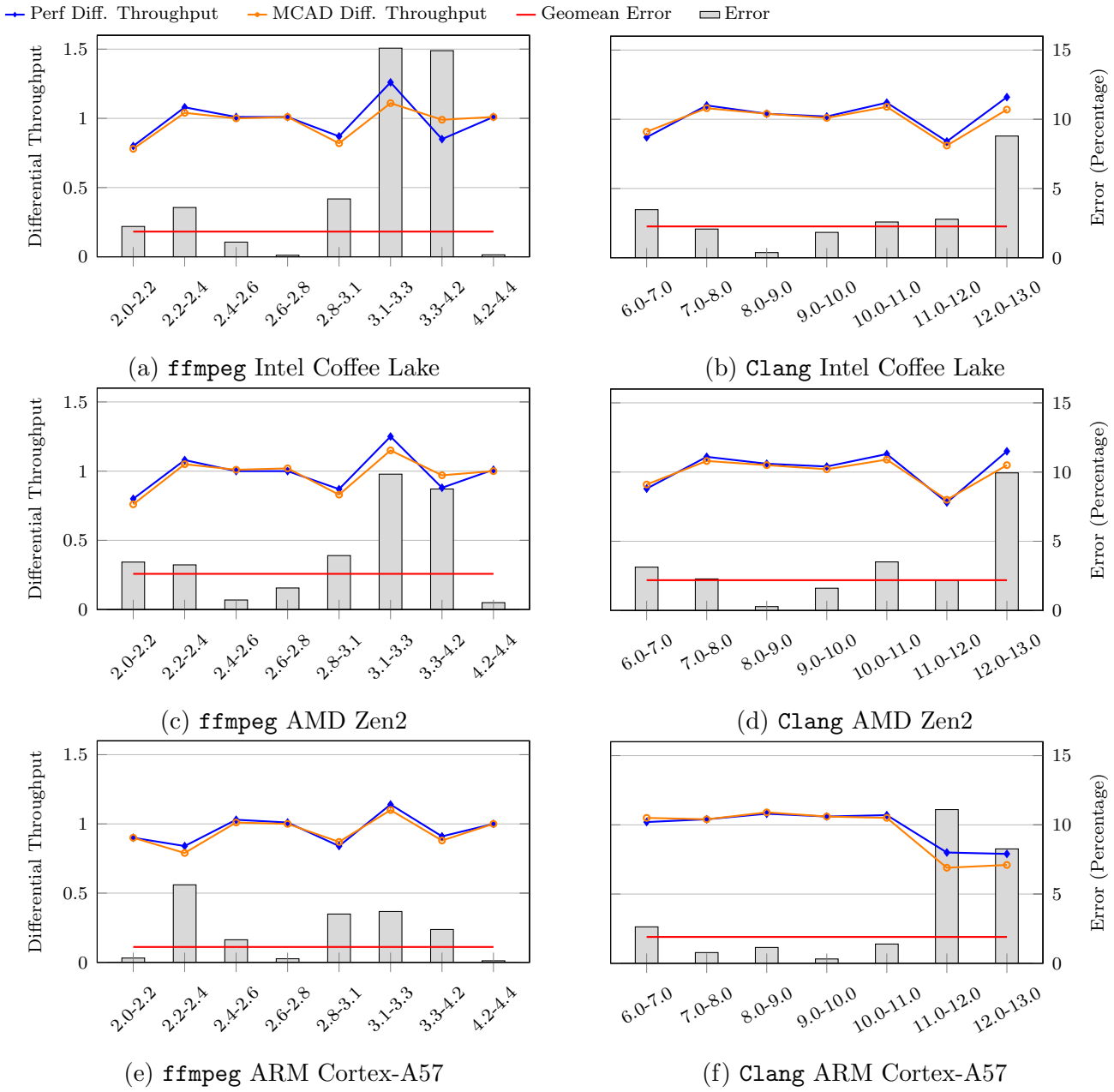
To assess the overall accuracy and ability to generalize throughput predictions across control transfers we conduct experiments using two popular and widely used applications as target

programs: the `ffmpeg` video encoder/decoder and the C/C++/Objective-C compiler `clang`. We select `ffmpeg` for our case study as video encoding represents a complex and highly performance-intensive task with many applications in real-world use cases. `clang` represents a large-scale software consisting of complex branching logic, which is well suited to test MCAD's cross-branch prediction accuracy and also plays an important role in many real-world scenarios.

We collect baseline cycle-count measurements on three physical machines with different instruction set architectures (ISAs) and microarchitectures:

- *Intel CoffeeLake:* 6-core Intel i7 8700K x86_64 CPU, clocked at 3.70GHz and 32G of RAM running Ubuntu 20.04
- *AMD Zen 2:* 12-core AMD Ryzen 9 3900X x86_64 CPU, clocked at 2.48GHz and 32G of RAM running Ubuntu 20.04
- *ARM Cortex-A57:* 4-core ARM Cortex-A57 AArch64 CPU, clocked at 1.73GHz and 4G of RAM running Ubuntu 16.04

Compared to running baseline measurements on smaller program scopes (*e.g.* a single basic block), measuring larger execution traces faces much more operating system noise. To avoid noise due to CPU migration or context switching we allocate a single processor core exclusively for the process under measurement. In addition, we disable Simultaneous Multithreading (SMT, also called *Hyper-Threading* on Intel processors) on the benchmarking core since it is not supported by our analysis engine as mentioned in Section 4.3.5. The baseline measurements are obtained using Linux Perf, which leverages the Performance Monitor Unit (PMU) provided by the underlying hardware, and are averaged over 1000 repetitions.



(a) `ffmpeg` Intel Coffee Lake

(b) `Clang` Intel Coffee Lake

(c) `ffmpeg` AMD Zen2

(d) `Clang` AMD Zen2

(e) `ffmpeg` ARM Cortex-A57

(f) `Clang` ARM Cortex-A57

Figure 4.4: Differential execution timing comparisons between subsequent development versions of `ffmpeg` and `clang`. Relative cycle count differences are plotted in blue and orange for MCAD and Perf respectively. The gray bars represent the Mean Absolute Percentage Error for each experiment, the geometric mean of all error values is plotted in red. In all experiments, we compare MCAD's predictions against hardware-performance counter measurements collected from execution traces on the respective physical devices.

FFmpeg

We evaluate `ffmpeg` on subsequent version pairs using 9 different release versions in the following order: 2.0, 2.2, 2.4, 2.6, 2.8, 3.1, 3.3, 4.2, and 4.4. For each experiment, we use the same 14KB MPEG-4 video file as reference input and execute the following command:

```
ffmpeg -i input.mp4 -f null -
```

Figure 4.4a, 4.4c, and 4.4e depict the differential throughput predictions of MCAD, between

$$\Delta_{MCAD}(\texttt{ffmpeg}_i,\ \texttt{ffmpeg}_j)$$

and the baseline Δ_{ij} for version pairs (i, j) as $(2.0, 2.2)$, $(2.2, 2.4)$, and so forth. In addition, the Figures present the resulting error rates E_{ij} for the corresponding version pairs between ground truth measurements on physical devices and predictions by MCAD.

Our results show that MCAD closely follows hardware cycle counts and never deviates from changes in the baseline count by more than 15%. On average, the error is 1.8% for Intel, 2.6% for AMD, and 1.1% for the ARM Cortex-A57 with a standard deviation of less than 6.3% in all cases. Nevertheless, on both Intel and AMD machines, two version pairs show unusually high error: $(3.1, 3.3)$ and $(3.3, 4.2)$ deviate from baseline measurements by around 10% to 15%. Further investigation showed that this deviation is likely due to a high number of cache misses during the execution of `ffmpeg` version 3.3 on Intel and AMD machines. Figure 4.5 shows the number of cache misses when running different versions of `ffmpeg` on CoffeeLake and Zen 2, measured using hardware performance counters averaged over 1000 repetitions. On both machines the number of cache misses spikes for version 3.3, exceeding the second highest measurement by at most 30%. As detailed in Section 4.3.2, MCAD utilizes LLVM's instruction scheduling database for instruction latency information which assumes that all memory accesses result in cache hits. Therefore, without proper and potentially

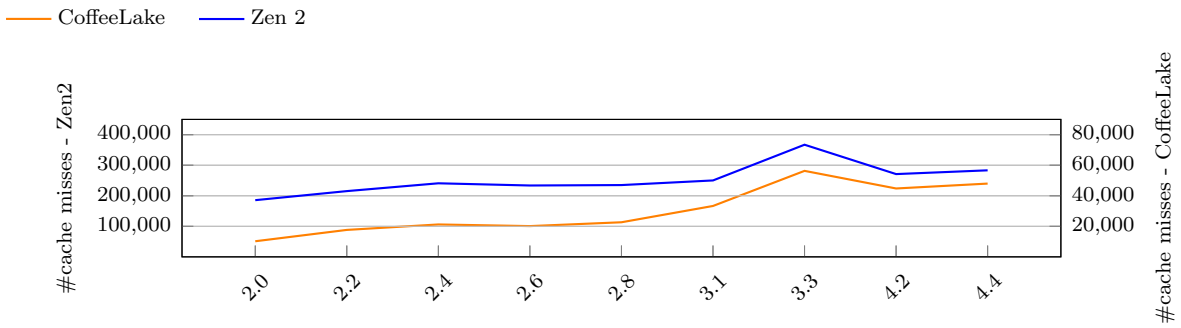


Figure 4.5: Number of cache misses during the execution of different `ffmpeg` versions on CoffeeLake and Zen2 machines.

expensive cache simulation (See Section 4.3.5), MCAD's precision will be hindered by cycle count penalties originating from cache misses.

Clang

We conducted our experiments for `clang` using 8 different release versions as follows: 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, and 13.0. In order to reduce the amount of I/O interference and simplify the experiments without losing generality, we focus on the backend of `clang`'s compilation pipeline. More specifically, we measure the cycles consumed by `clang` in compiling an unoptimized LLVM IR program to an object file. The LLVM IR input file (`input.ll`) is generated from the following C program:

```
int foo(int x, int y) {
  return x * 2 + y;
}
```

We use `input.ll` as the reference input for our experiments and execute the following command:

```
clang -O2 -c input.ll -o /dev/null
```

Figure 4.4b, 4.4d, and 4.4f depict the differential throughput predictions of MCAD, between

$$\Delta_{MCAD}(\texttt{clang}_i,\ \texttt{clang}_j)$$

and the baseline Δ_{ij} as well as the error rates E_{ij} between predictions by MCAD and baseline for version pairs (i, j) as $(6, 7)$, $(7, 8)$, and so forth.

Again, our results show that MCAD closely follows the hardware cycle count and never deviates from the changes in the baseline count by more than 12%. On average, the error is 2.3% for Intel, 2.2% for AMD, and 1.9% for the ARM Cortex-A57 with a standard deviation of less than 5% in all cases. On both Intel and AMD machines, we observe a spike of nearly 10% error on version pair $(12.0, 13.0)$. Similar to the culprit for unusually high error percentage in Section 4.5.1, we find that this spike of error is caused by higher number of cache misses on version 13.0: compared to other `clang` versions, `clang` 13.0 creates 13% to 35% more cache misses on both machines. In addition, on the ARM machine, we observe around 10% of error on version pairs $(11.0, 12.0)$ and $(12.0, 13.0)$. We find that this is caused by sudden increase of memory operations in both version pairs. For instance, the number of `LDUR` (memory load) and `STR` (memory store) instructions increases by a factor of 27 in said version pairs. Modern processors usually apply advanced optimizations, such as load-store forwarding, during the execution of these memory operations which might not be accurately modeled by MCA.

Version Delta	# LoCC
6.0 ∼7.0	3,040,664
7.0 ∼8.0	2,699,505
8.0 ∼9.0	3,099,259
9.0 ∼10.0	2,676,054
10.0 ∼11.0	3,795,173
11.0 ∼12.0	6,029,487
12 .0 ∼13.0	5,629,956
Geomean	3,658,380.7

Table 4.1: # LoCC between clang release versions

Version Delta	# LoCC
14.0.0 ∼14.0.1	2,761
14.0.1 ∼14.0.2	4,515
14.0.2 ∼14.0.3	76
14.0.3 ∼14.0.4	3,151
14.0.4 ∼14.0.5	1,453
14.0.5 ∼14.0.6	596
Geomean	1,171.5

Table 4.2: # LoCC between clang minor release versions

Version Delta	# LoCC
2.0 ∼2.2	205,518
2.2 ∼2.4	197,301
2.4 ∼2.6	128,982
2.6 ∼2.8	226,955
2.8 ∼3.1	274,626
3.1 ∼3.3	193,774
3.3 ∼4.2	390,620
4.2 ∼4.4	293,100
Geomean	227,723.0

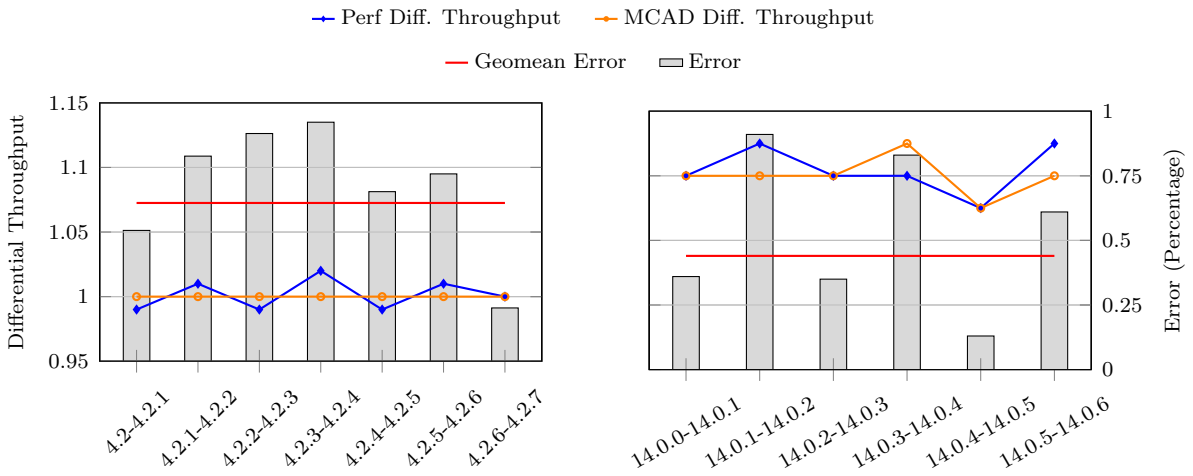
Table 4.3: # LoCC between ffmpeg release versions

Version Delta	# LoCC
4.2 ∼4.2.1	482
4.2.1 ∼4.2.2	2,013
4.2.2 ∼4.2.3	1,687
4.2.3 ∼4.2.4	773
4.2.4 ∼4.2.5	2,098
4.2.5 ∼4.2.6	332
4.2.6 ∼4.2.7	132
Geomean	735.4

Table 4.4: # LoCC between ffmpeg minor release versions

4.5.2 Differential Throughput Prediction on Small Changes

So far, MCAD has shown its accurate differential throughput predictions on changes between major software release versions. In this sub-section, we futher illustrate MCAD's capability of predicting differential throughput originating from (much) smaller changes, like software patches. This is supported by repeating the Intel CoffeeLake experiments from Section 4.5.1 and Section 4.5.1, but using *minor* `ffmpeg` and `clang` releases, which have far less modifi-



(a) `ffmpeg` (b) `Clang`

Figure 4.6: Differential throughput prediction of small software changes in `ffmpeg` and `Clang` on Intel CoffeeLake

cations between them, rather than their major releases as the analysis targets. Ideally, we should characterize the size of changes between two different versions based on their program binaries. However, measuring differences between two binaries is a hard problem [22, 50]. Therefore, we use lines of (source) code changes (# LoCC) as an approximate metric of change between two software versions.

For our evaluation, we repeat the experiments detailed in Section 4.5.1 for `ffmpeg` on 7 minor releases for version 4.2: 4.2.1, 4.2.2, 4.2.3, 4.2.4, 4.2.5, 4.2.6, and 4.2.7. The # LoCC between minor releases range from 132 to around 2000 lines with a geomean of 735 lines (See Table 4.4), whereas # LoCC between major `ffmpeg` releases evaluated in Section 4.5.1 range from 128k to 390k lines with a geomean of 227k (See Table 4.3). Figure 4.6a shows the differential throughput results along with their error percentage.

We repeat the experiment for `clang`, for which we evaluate 6 minor 14.0 releases: 14.0.1, 14.0.2, 14.0.3, 14.0.4, 14.0.5, and 14.0.6. The # LoCC between these minor releases range from 76 to around 4k lines with a geomean of 1171 lines (See Table 4.2). In contrast, the # LoCC between major `clang` releases evaluated in Section 4.5.1 range from 2.6M to

6M lines with a geomean of 3.6M lines (See Table 4.1). Figure 4.6b shows the differential throughput results as well as the respective their error percentage.

Both experiments show that MCAD's predictions follow the hardware cycle counts closely, with only marginal error rate of less than 1% on average and 1.5% in the worst case.

4.5.3 Scalability and Comparison

Besides accurate predictions that generalize across control-flow transfers, another major goal of MCAD is scalability. In particular, we aim for MCAD to scale up with the complexity of real-world target programs. In this section, we compare against four state-of-the-art throughput prediction and analysis approaches: OSACA [35], Ithemal [43], uiCA [6], and LLVM-MCA [2]. We present the results in Table 4.5.

First, we focus on the benchmarks these tools used in their repositories or publications. We compare the type and size of benchmark, as well as their supported target instruction sets. All prior art operates on the individual basic block level with the exception of LLVM MCA which also contains designated support for loop kernels. However, in both cases instruction sequences usually consist of only 10∼20 instructions at most. On the other hand, MCAD was designed to work on real-world program traces containing *millions* of instructions. MCAD also supports most of the hardware architectures that QEMU and LLVM support, which amounts to nearly 20 different Instruction-Set Architectures (ISAs). In contrast, most other tools are highly architecture specific and only support x86_64.

We further evaluate these tools using the same reference input and compare their performance with respect to execution time and memory consumption. For this purpose, we collected the execution trace of a `ffmpeg` invocation using version 4.2 as described in Section 4.5.1 storing the results into a file. The resulting instruction stream consists of roughly 27 million x86_64

	uiCA [6]	OSACA [35]	Ithemal [43]	LLVM MCA [2]	**MCAD**
execution time	Timeout after 48h.	Exit w/error after 24h.	Exit w/error after 2m.	219.98s	**52.69s**
`ffmpeg` results	✗	✗	✗	✓	✓
`clang` results	✗	✗	✗	✓	✓
memory usage	113GB	N/A	N/A	29.39GB	**2.16GB**
scales to # of instrs.	10∼20	∼40	10∼20	∼1000	**>1000000**
mean error	**3%** [6]	∼30% [6]	∼5% [6]	∼20% [6]	**3%**
benchmark type	basic block	loop kernel	basic block	loop kernel	**whole program**
supported ISAs	x86_64 only	x86_64 only	x86 & x86_64	**∼20 ISAs**	**∼20 ISAs**
handles branches	✗	✗	✗	✗	✓

Table 4.5: Comparison of different tools to predict cycle counts of software across various dimensions. While the error metrics differ, we present error numbers as reported by the most recent work [6] for completeness.[5]

instructions. To give state-of-the-art approaches the benefit of the doubt we perform this experiment on an 80-core Intel Xeon E7-4870 machine, clocked at 2.4GHz, equipped with 198GB of RAM and the same amount of swap space, setting a 48-hour time limit on the execution.

As shown, OSACA and Ithemal did not finish this task: OSACA bailed out with failures related to loading hardware models after parsing the input file; Ithemal promoted an out-of-memory error from its DynamoRIO [16] runtime before bailing out. Similarly, uiCA could not finish within the time limit after consuming significant amount of memory. Last but not the least, despite being able to finish, LLVM-MCA consumed significantly more time and memory compared to MCAD, demonstrating the effectiveness of our changes over standard MCA in our implementation.

4.6 Discussion

Current state-of-the-art throughput prediction approaches either explicitly model or learn microarchitectural implementation details, resource usage, instruction scheduling, and latency numbers, using data. This data is sometimes provided by processor vendors directly, although, most of the time it is collected using emperical methods, like measuring instruction latencies using many different software configurations, and large numbers of repetitions to reduce inherent error signals.

MCAD builds on top of this prior work that provides insights into modern processor pipelines through detailed measurements and experiments. Since a lot of this research has been contributed in part by vendors directly and in other parts incorporated by the community into the LLVM compiler infrastructure, MCAD currently uses LLVM MCA as the core analysis

[5]uiCA uses the Mean Absolute Percentage Error (MAPE) to compare the error of a prediction against a single execution on a physical device, whereas we use the mean error of the predicted *difference* in cycles between two executions.

engine. However, the core analysis component in our design can support other throughput analysis engines in principle, which would allow us to predict timing effects of a number of optimizations in modern processors that are not currently modeled by LLVM MCA, such as including instruction prefetching and branch prediction which usually happen in the processor frontend. The main obstacle towards that as demonstrated by our experiments in Section 4.5.3, however, remains overcoming scalability issues of the related approaches.

Looking ahead to future work we anticipate that research into analysis of multi-process and multi-thread executions should be feasible within MCAD in principle. As introduced in Section 4.3 we collect execution traces using QEMU and a custom plugin and certain recently-added QEMU plugin interfaces would allow us to distinguish traces originating from different virtual CPUs at runtime. Nevertheless, how to incorporate modern processors' concurrency models into current throughput prediction approaches remains an open research question. It would also be possible to substitute QEMU for other methods of execution trace collection entirely: for instance, leveraging binary rewriting tools would enable us to insert instrumentations that report the executed instructions natively.

Last but not least, we believe a more scalable, intuitive, and interactive timeline or waterfall view could provide developers with more insights by visualizing resource dependencies among instructions, pointing towards potential avenues for improving continuous development of timing-sensitive code.

4.7 Summary

To summarize, our results show that MCAD improves on the state of the art by scaling up to complex real-world software. It is well suited to providing cycle count estimates with rapid developer-centric turn-around times while targeting a range of different hardware

architectures. MCAD can drive software development by quickly iterating on small changes and assessing their timing impact on real-world programs such as `ffmpeg` and `clang`, with a mean error in differential throughput estimates of $< 3\%$ compared to hardware-based measurements.

Chapter 5

Related Works

5.1 Improvements on the Scalability of IFDS

The original IFDS/IDE algorithm [51] together with practical extensions proposed by Naeem et al. [45] is nowadays implemented by many analysis frameworks for JAVA [14, 10] and C/C++ [53]. In addition, several approaches have been proposed to further reduce memory consumption and processing time of IFDS implementations. Sparsedroid [28] improves the sparsity of the dataflow propagation. The number of dataflow edges is reduced by connecting dataflow facts directly to their next point of use, instead of the next node in the CFG. DiskDroid [37] and CleanDroid [9] reduce the footprint of the graph reachability algorithm by detecting stale edges and either move them to disk or completely remove them from the working set. Coyote [56] improves the parallelism of bottom-up IFDS implementations by increasing the granularity of caller-callee dependencies. Intraprocedural analysis is split into multiple independent parts which can then be run in parallel. All of the mentioned approaches address shortcomings that are specific to the original IFDS implementation and are therefore not directly comparable to DFI.

5.2 Sparse Value-Flow Analysis

SVF [57] is another sparse value-flow analysis framework focused on computing points-to information. Similar to DFI, SVF operates on the sparse SSA def-use chains. To capture the relations between address taken variables SVF uses its own MemorySSA constructions derived from precomputed points-to information, like Andersen's analysis [8], and continuous refinements on the sparse value flow graph (SVFG). Different from DFI and IFDS based approaches, SVF does not support the configuration of custom value-flow transfer functions.

5.3 Program Analysis via Big Data analytics

Recent contributions such as BigSpa [66], Grapple [67], GraSpan [62] and Chianina [68] work around scalability issues of static analysis frameworks by developing core functionalities inspired by big data analytics in order to support certain classes of static analyzers. The actual analysis is then implemented on top of the core API and can thereby be transparently scaled to the available resources of the underlying system. Systems approaches are orthogonal to DFI as they aim to provide primitives to improve the resource utilization of static analyzers but do not aim to optimize the analysis algorithm itself.

5.4 Static Timing Analyses

A large body of prior research focused on static prediction of worst-case timing behavior [54, 39, 23, 40, 27]. However, reasoning about timing properties of arbitrary programs reduces to the halting problem in the general case and as a result approaches for calculating worst case execution time make strong assumptions such as an upper bound on the number of

loop iterations, recursion depth, effects of memory accesses, and external I/O operations. In practice, this means that the user of traditional tools has to provide upper bound information for all loop constructs, recursion, avoid indirect memory accesses through pointers, and avoid the use of I/O operations in analyzed parts of the code. Ensuring proper and correct usage then typically requires dedicated build toolchains and environment setups, as well as expert knowledge about the analysis framework. Moreover, such tools typically over-approximate cycle counts up to several orders of magnitude over physical hardware execution in order to remain sound, with several tools providing timing estimates in units of *wall-clock time* rather than cycles [44, 7].

More recently, a number of approaches [6, 35, 2, 43] proposed throughput modeling of machine code using parametric models for accurate, yet fast throughput prediction. Such approaches predict timing aspects of a particular instruction sequence of the target program rather than reasoning about the entire set of possible executions at once like prior static approaches do. While their underlying parametric models require knowledge of key microarchitectural aspects such as port usage, instruction latencies, and other internal details that may not be publicly available, recent advances in machine learning showed that architecture dependence can be tackled to some extent by *learning* model parameters from data [43]. However, without ruling out the use of learning-based solutions, our evaluation show that currently such approaches are severely limited with respect to scalability, providing throughput estimates only for a handful up to a few hundred instructions at most, also lacking support for prediction across control-flow transfers. In contrast, MCAD handles complex binary programs containing literally millions of individual instructions with near-native execution speeds.

5.5 Dynamic Timing Analyses

Dynamic approaches aim at providing detailed and concrete timing analyses of the running software using concrete inputs. There are two main flavors of dynamic timing analysis tools: either using physical hardware tracing or using architectural simulators. Approaches using physical tracing execute the program on the target architecture and measure cycle counts directly using the facilities provided by the device [41, 3]. While in theory this yields the most precise results and should also be reasonably fast, in practice this is often not the case: the target architecture might be a production system that is not readily available to the developer running the test and in a collaborative environment each team would require their own physical device to test their changes against.

Additionally, setting up and using facilities for accurate cycle-count measurements can be a time-intensive task in and of itself, requiring complicated setup, and potentially support by the target program's build toolchain as well as the operating system of the production system. Worse yet, the target architecture might not actually provide any built-in facilities for accurate measurement of cycle counts, requiring developers to implement purpose-built, custom, and highly architecture-dependent in-house measurement frameworks, whose accuracy might actually be limited in the end.[1] Cycle-accurate architectural simulators [17, 65, 13] on the other hand promise to provide a similar level of accuracy as physical tracing without requiring an actual physical device to capture program execution. Unfortunately, simulation-based approaches also come with major drawbacks: first, performance is typically at least three orders of magnitude slower than native execution (or even slower) as they faithfully simulate microarchitectural details of modern processor pipelines completely in software. Second, they are usually aimed towards explorative hardware design and implementation studies of novel architectures rather than simulating throughput of software for existing platforms. As

[1] For instance, because of requiring *instrumentation* of the original binary where instrumentation overhead cannot easily be measured at runtime.

a result, these frameworks are not easily accessible and can be difficult to integrate with existing software development tools and continuous integration workflows due to the high resource requirements and time-intensive nature of the simulation-based approach.

Chapter 6

Conclusion

Software codebases are getting larger and more complex, but contemporary program analysis has yet kept up with them. The inability to scale computation resources used by analyses with these large codebases often leads to reduced analysis quality and hinders the crucial tasks that are depending on these results, such as bug finding and performance tuning. In this dissertation we show some ideas and paths to solve this issue in dataflow and throughput analysis, two of the most important program analysis sub-fields.

Dataflow analysis plays a central role in programming language and software engineering fields. People have proposed countless of ideas to improve the scalability of dataflow analysis on different problems. For instance, IFDS-based solver and adopting sparse representation for program under analysis. However, there is yet a solution to leverage both the aforementioned techniques in a coherent way. Our efficient value-flow framework for IFDS problems, DFI, not only combines traditional IFDS framework with LLVM-IR-based sparse program representation, but more importantly, creates a novel depth-first, interval-based graph reachability algorithm that traverses the SSA graph in *reverse* direction, as reverse SSA graph has a relatively low tree width hence suitable for efficient tree-based solutions. All together,

DFI scales better than sate-of-the-art IFDS framework on large codebases like OpenSSL by several magnitudes, with comparable precision.

Throughput analysis is key to providing insights to the development of many performance sensitive applications. Unfortunately, contemporary throughput analysis techniques that use either static or dynamic approach struggle to strike a balance between performance and precision, which often leads to difficulties in analyzing non-trivial workload from real-world software. MCAD, our hybrid throughput analysis framework, combines the advantages of static and dynamic strategies by streaming program execution traces from performant emulators to efficient static throughput analyzer. MCAD scales well with large codebases like FFmpeg and Clang, outperforming other state-of-the-art static throughput analyzers. In terms of precision, our framework also achieves less than 3% geo. mean error compared to ground truth timings on the task of differential throughput analysis targeting x86 and ARM machines.

Bibliography

[1] Chrome devtools. `https://developer.chrome.com/docs/devtools/`. Accessed: 2021-12-13.

[2] Llvm mca. `https://llvm.org/docs/CommandGuide/llvm-mca.html`. Accessed: 2021-06-28.

[3] Perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 2021-11-08.

[4] Intro to the llvm mc project. `https://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html`, 2010. Accessed: 2021-07-08.

[5] A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[6] A. Abel and J. Reineke. Accurate throughput prediction of basic blocks on recent intel microarchitectures. *arXiv preprint arXiv:2107.14210*, 2021.

[7] J. Abella, C. Hernández, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-Askasua, J. Perez, E. Mezzetti, and T. Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015.

[8] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.

[9] S. Arzt. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pages 1098–1110, Madrid, Spain, May 2021. IEEE Press.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, Edinburgh United Kingdom, June 2014. ACM.

[11] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 553–566, 2015.

[12] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. In *ACM SIGARCH computer architecture news*, volume 39, pages 1–7, 2011.

[14] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, June 2012. Association for Computing Machinery.

[15] A. Browne and J. Zhao. Dataflowsanitizer, Accessed: 2022.

[16] D. Bruening and S. Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[17] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. In *ACM SIGARCH computer architecture news*, volume 25, pages 13–25, 1997.

[18] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[19] F. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *International Conference on Compiler Construction*, pages 253–267. Springer, 1996.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[21] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[22] Y. Duan, X. Li, J. Wang, and H. Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and Distributed System Security Symposium*, 2020.

[23] H. Falk and J. C. Kleinsorge. Optimal static wcet-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference*, pages 732–737, 2009.

[24] N. Grech and Y. Smaragdakis. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[25] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices*, 44(1):226–238, 2009.

[26] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, 2011.

[27] D. Hardy, B. Rouxel, and I. Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[28] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279, Nov. 2019.

[29] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 594–601, 2005.

[30] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta informatica*, 7(3):305–317, 1977.

[31] J. Kodumal and A. Aiken. The set constraint/cfl reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.

[32] D. C. Lars Rasmusson. Performance overhead of kvm on linux 3.9 on arm cortex-a15. `https://www.diva-portal.org/smash/get/diva2:1043325/FULLTEXT01.pdf`, 2013.

[33] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[34] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[35] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*, pages 121–131. IEEE, 2018.

[36] J. Lee and A. Shrivastava. Static analysis of register file vulnerability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):607–616, 2011.

[37] H. Li, H. Meng, H. Zheng, L. Cao, J. Lu, L. Li, and L. Gao. Scaling up the IFDS algorithm with efficient disk-assisted computing. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 236–247, Virtual Event, Republic of Korea, Feb. 2021. IEEE Press.

[38] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 343–353, 2011.

[39] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. In *Science of Computer Programming*, volume 69, pages 56–67, 2007.

[40] B. Lisper. Sweet–a tool for wcet flow analysis. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer, 2014.

[41] R. LTD. Automating wcet analysis for do-178b/c. 2017.

[42] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, 2000.

[43] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

[44] E. Mezzetti and T. Vardanega. *On the industrial fitness of wcet analysis*. na, 2011.

[45] N. A. Naeem, O. Lhoták, and J. Rodriguez. Practical Extensions to the IFDS Algorithm. In R. Gupta, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 124–144, Berlin, Heidelberg, 2010. Springer.

[46] D. Novillo. Design and implementation of tree ssa. In *Proceedings of GCC developers summit*, pages 119–130. Citeseer, 2004.

[47] D. Novillo et al. Memory ssa-a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers' Summit*, pages 97–110. Citeseer, 2007.

[48] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for c-like languages. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 229–238, 2012.

[49] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 104–118, 1977.

[50] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 142–157, 2021.

[51] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[52] N. Robertson and P. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[53] P. D. Schubert, B. Hermann, and E. Bodden. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 393–410, Cham, 2019. Springer International Publishing.

[54] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static wcet analysis of real-time task-oriented code in vehicle control systems. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 212–219. IEEE, 2006.

[55] K. Serebryany and M. Elver. Sanitizercoverage, Accessed: 2022.

[56] Q. Shi and C. Zhang. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 835–847, Seoul South Korea, June 2020. ACM.

[57] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[58] S. Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[59] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[60] D. Vardoulakis and O. Shivers. Cfa2: A context-free approach to control-flow analysis. In *European Symposium on Programming*, pages 570–589. Springer, 2010.

[61] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

[62] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. *ACM SIGPLAN Notices*, 52(4):389–404, Apr. 2017.

[63] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 154–164, 2014.

[64] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[65] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, 2007.

[66] Z. Zuo, R. Gu, X. Jiang, Z. Wang, Y. Huang, L. Wang, and X. Li. BigSpa: An Efficient Interprocedural Static Analysis Engine in the Cloud. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 771–780, May 2019.

[67] Z. Zuo, J. Thorpe, Y. Wang, Q. Pan, S. Lu, K. Wang, G. H. Xu, L. Wang, and X. Li. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, New York, NY, USA, Mar. 2019. Association for Computing Machinery.

[68] Z. Zuo, Y. Zhang, Q. Pan, S. Lu, Y. Li, L. Wang, X. Li, and G. H. Xu. Chianina: An evolving graph system for flow- and context-sensitive analyses of million lines of C code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 914–929, New York, NY, USA, June 2021. Association for Computing Machinery.