# Understanding Program Structure and Behavior

by

Sie Hendrata Dharmawan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006



Author .....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by .....
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

**Understanding Program Structure and Behavior**

by

Sie Hendrata Dharmawan

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

## Abstract

A large software system usually has structure in it. Several functions work together to accomplish a certain task, and several tasks are grouped together to perform a bigger task. In order to understand this division, one has to consult the documentations or read through the source code. However, the documentations are usually incomplete and outdated, while code inspection is tedious and impractical. Algorithms have been proposed that automatically group functions with similar functionality.

In this thesis I will present LogiView, an algorithm that presents an organizational view of the functions. This view will ease the process of understanding the structures in the program, identifying functions with related tasks, and separating the functions into logical groups.

I will also present a methodology of analyzing the function names in the program. This method leverages the result of the LogiView algorithm and identifies the names that are most relevant to the functionality of the program. Given a set of programs that are known to have same functionality, this method extracts the similarity in the function names and builds a dictionary of the names that are semantically related to the functionality. The methodology also detects when two programs have similar functionality and how to measure the similarity between multiple programs.

Lastly, I will show that the result of the LogiView algorithm can be used to explain program performance. The abrupt change in the program behavior and repetitions of past behaviors can be explained in terms of function calls, and these functions are identified by the LogiView algorithm.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

# Acknowledgments

I would like to thank Saman Amarasinghe as my thesis supervisor, and also Rodric Rabbah for guiding me and providing me with ideas throughout the experiments and the writing of this Thesis.

This work was supported in part by NSF award CNS-0305453.

# Contents

**1 Introduction** **15**

1.1 Contributions . . . . . 17

1.2 Thesis Roadmap . . . . . 19

**2 Background and Related Work** **21**

2.1 Software Oriented Approaches . . . . . 21

2.2 Hardware Oriented Approaches . . . . . 24

**3 LogiView Algorithm** **27**

3.1 Overview . . . . . 27

3.2 Call Graph . . . . . 27

3.3 Dominator Tree . . . . . 28

3.4 Shared Node . . . . . 33

3.5 Pathological Patterns . . . . . 36

3.5.1 Chain Pattern . . . . . 36

3.5.2 Leafy Pattern . . . . . 37

3.6 Finding the Phase-Markers . . . . . 37

**4 Experiments and Analysis** **43**

4.1 Experiments and Results . . . . . 44

4.1.1 Timberwolf Benchmark . . . . . 44

4.1.2 Link Grammar Parser . . . . . 51

4.1.3 GNU Zip . . . . . 55

4.1.4 Versatile Place and Route . . . . . 57

**5 Function Names Analysis** **65**

5.1 Overview . . . . . 66

5.2 Name Splitting . . . . . 67

5.3 Application Profile . . . . . 68

5.4 Class Dictionary . . . . . 70

5.5 Dictionary Construction . . . . . 70

5.6 Punishing Generic Names . . . . . 71

5.7 Application Score . . . . . 73

5.8 Application Class Analysis . . . . . 75

5.8.1 Satisfiability Solver . . . . . 75

5.8.2 Support Vector Machines Learning . . . . . 77

5.8.3 Deterministic Finite Automata Learner . . . . . 78

5.8.4 Linear Programming Solver . . . . . 80

5.8.5 Fast Fourier Transform . . . . . 82

5.8.6 Punishment Function . . . . . 84

5.8.7 Leave-One-Out Test . . . . . 84

**6 Physical Phase Detection** **89**

6.1 Results . . . . . 92

6.1.1 SVM Light . . . . . 92

6.1.2 SPEC Gnu Zip . . . . . 98

6.1.3 Timberwolf Benchmark . . . . . 101

6.1.4 Grammar Link Parser . . . . . 104

6.2 Correlation to Phase Detection Problem . . . . . 109

**7 Discussion and Conclusion** **113**

7.1 Future Research . . . . . 114

# List of Figures


3-1 An algorithm that outputs the dominators of each node [26] pp. 182. 29

3-2 A section of the call graph from zChaff benchmark. . . . . 30

3-3 The dominator tree from zChaff benchmark. . . . . 31

3-4 A section of the dominator tree from zChaff benchmark. . . . . 31

3-5 The pseudocode of the zChaff benchmark. . . . . 32

3-6 A sample program that would cause a shared node of the second kind. `analyze_conflicts` is shared between `init_solve` and `deduce`. . . . 34

3-7 A section of the call graph and dominator tree that would result from Figure 3-6. . . . . 34

3-8 A delegation of task that causes a chain pattern in the dominator tree. 36

3-9 The dominator tree formed by the code in Figure 3-8. . . . . 36

3-10 An algorithm that finds the phase-markers in the dominator tree. . . 39

3-11 A hypothetical dominator tree. . . . . 40

4-1 The pseudocode of the Timberwolf Benchmark. . . . . 45

4-2 The simplified view of the Timberwolf code. . . . . 46

4-3 A snippet from `utemp.c` in the Timberwofl Benchmark. . . . . 47

4-4 The call graph for the Timberwolf Benchmark. . . . . 48

4-5 The dominator tree of the Timberwolf Benchmark. . . . . 48

4-6 A snippet from `uloop.c`, `ucxx1.c`, and `ucxx2.c` in the Timberwofl Benchmark. . . . . 50

4-7 The original pseudocode for the Link Grammar Parser. . . . . 51

4-8 The call graph for the Link Grammar Parser. . . . . 52

4-9 The dominator tree of the Link Grammar Parser. . . . . 52
4-10 The subtree rooted in `read_dictionary` in the SPEC parser benchmark. 53
4-11 The subtree rooted in `build_sentence_disjuncts` in the SPEC parser benchmark. . . . . 53
4-12 The subtree rooted in `power_prune` in the SPEC parser benchmark. . 54
4-13 The call graph of the GNU Zip Benchmark. . . . . 55
4-14 The dominator tree of the GNU Zip Benchmark. . . . . 56
4-15 The Negotiated Congestion algorithm [23]. . . . . 58
4-16 The Dijkstra's Algorithm to find the shortest path from $s$ to $t$ in a directed graph $G$. . . . . 59
4-17 The call graph for the VPR. . . . . 60
4-18 The dominator tree of the VPR Benchmark. . . . . 60
4-19 The subtree dominated by `route_net` in the VPR benchmark. . . . . 61
4-20 A partial snippet of the `route.c` file in the VPR Benchmark, showing the main steps of the `route_net` function. . . . . 61
4-21 The subtree dominated by `try_route` in the VPR benchmark, omitting the subtree dominated by `build_rr_graph` and `route_net`. . . . . 62
4-22 A partial snippet of the `route.c` file in the VPR Benchmark, showing the main steps of the `try_route` function. . . . . 63

5-1 A section of the dominator tree from the zChaff benchmark. . . . . 66
5-2 A section of the dominator tree from the zChaff benchmark. . . . . 66

6-1 Miss rates plot of a benchmark where there are higher miss rates region and lower miss rates region. . . . . 90
6-2 A section of miss rates plot where a similar behavior is repeated multiple times. . . . . 90
6-3 The entry and exit to `optimize_to_convergence` in the SVM Light Benchmark. . . . . 92
6-4 Multiple entries and exits to `optimize_svm` in the SVM Light Benchmark. 93

6-5 A zoomed-in view of Figure 6-4, showing the repetition of the pattern for each call to `optimize_svm`. . . . . 93
6-6 A simplified view of the SVM Light code. . . . . 94
6-7 A section of the dominator tree under `optimize_to_convergence`, only showing the child `optimize_svm`. . . . . 95
6-8 A section of the dominator tree under `optimize_to_convergence`, omitting the child `optimize_svm`. . . . . 96
6-9 A slightly zoomed-in view of Figure 6-5, with the entries and exits to `update_linear_component`. . . . . 97
6-10 A slightly zoomed-in view of Figure 6-5, with the entries and exits to `select_next_qp_subproblem_grad`. . . . . 97
6-11 Entries to and exits from `spec_compres` in the gzip benchmark. . . . 98
6-12 Entries and exits to `deflate` in the gzip benchmark. . . . . 99
6-13 Entries to and exits from `flush_block` in the gzip benchmark. . . . . 100
6-14 A zoomed in view of Figure 6-13, showing a repeated pattern between two consecutive entries or exits from `flush_block`. . . . . 101
6-15 Entry and exit from `utemp` in the Timberwolf Benchmark. . . . . 102
6-16 A zoomed in view of Figure 6-15, showing the entry to `utemp` in the Timberwolf Benchmark. . . . . 102
6-17 Entries and exits from `uloop` in the Timberwolf Benchmark, showing two full repetitions. . . . . 103
6-18 The entire execution of the Grammar Link Parser. . . . . 104
6-19 Entries and exits to `read_entry` in the Grammar Link Parser. . . . . 105
6-20 Entry and exit to `prepare_to_parse` in the Grammar Link Parser. . . 105
6-21 Entries and exits to `power_prune` in the Grammar Link Parser. . . . 106
6-22 Entries and exits to `build_sentence_disjuncts` in the Grammar Link Parser. . . . . 107
6-23 Same plot as in Figure 6-10, but with $k = 32768$. . . . . 110
6-24 Same plot as in Figure 6-21, but with $k = 1024$. . . . . 111

# List of Tables


4.1 The phase-markers from the Timberwolf Benchmark and their parents. 44

4.2 The phase-markers from the Link Grammar Parser Benchmark and their parents. . . . . 51

4.3 The phase-markers from the GNU Zip Benchmark and their parents. 56

4.4 The phase-markers from the VPR Benchmark and their parents. . . . 59

5.1 The dictionary for the SAT application class. . . . . 76

5.2 The dictionary for the SVM application class. . . . . 77

5.3 The dictionary for DFA application class. . . . . 79

5.4 The dictionary for the LP application class. . . . . 81

5.5 The dictionary for FFT application class. . . . . 82

5.6 The punishment function for the shared tokens. . . . . 85

5.7 The similarity scores and the classification classes from the leave-one-out test. . . . . 86

6.1 The phase-markers from the GNU Zip Benchmark and their parents. 98

6.2 The phase-markers and phase-parents in identifying phase change or repeating pattern. . . . . 108

# Chapter 1

# Introduction

A large software system usually has structure in it. There are several modules who are responsible for certain aspects of the system, and within modules there are several classes and functions. Usually the best way to understand this structure is by inspecting the system documentation. But sometimes this straightforward approach is not possible because:

1. The documentation is not available.

2. The documentation only features the design of the system. The developers may implement it differently from what the designers intended. As a result, there are a lot of broken abstractions and violations of modularity, and the program structure is no longer obvious from the documentation.

3. As the system evolves with many revisions by different developers, the documentation does not list all the changes to the system. Moreover, the newer developers may change the design so the documentation no longer provides an accurate understanding of the program structure.

In the absence of documentation, it is still possible to understand the program structure by code inspection. But for large systems with hundreds of different functions and many lines of code, this approach is tedious and time-consuming. There have been some works in the past that attempts to automate the process of program

structure understanding to help guide programmers and system engineers as they navigate through unfamiliar codes. Some of these works even help programmers by suggesting program components. The components in a program are loosely defined as a set of codes whose task are closely related compared to the other components. For example, the components of a big software system may be IO classes, mathematical libraries, tree-traversal functions, etc. Some of the ideas to identify program components are:

1. As the program executes, its behavior gives a lot of information about program structure. For example, if the program behavior changes abruptly, it is usually a good indication that the program is performing a new task than it was before. By observing which functions are executed at different points and monitor changes in the program behavior, we can infer which functions are responsible for different tasks.

2. The relationships between different functions also provide information about program structure. When two functions are closely related to each other and responsible for the same task, it is likely that they are called one after another (i.e. they have strong temporal locality). Conversely, when we see that a call to function $A$ is always followed by a call to function $B$, we can speculate that these two functions are closely related and responsible for the same or similar task.

Based on these ideas, some works [5, 6, 13, 33, 44] attempted to infer program structure by measuring its observable physical characteristics. For example, they attempt to automatically detect when the observed characteristics have changed significantly, and signal that the program has entered a new *phase*. Thus, algorithms created with this approach are usually called phase detection algorithms. For example, the work of Dhodapkar and Smith [5, 6] monitors the last $N$ instructions that are used in the program (called the instruction working set). If this set changes abruptly between working set, then the program is executing a different set of instructions than before. This is a good indication that the program execution has moved to

a different component, and this new component's task is different than that of the previous component.

One problem with an approach that is based on observable physical characteristics is that it cannot describe and separate all the different program components. Observation of these measures can only detect when the program has entered a new phase, not give the description of the phase and relationships between different phases.

Another problem is that the resulting phases are dictated by their observable measurements, because the algorithms monitor the program and record the physical measurements as it executes. But in reality, the phases in the observable measurements are a result of executing the different components in the program, and not vice versa.

In this thesis, I will show that in order to understand program structure and behavior, one should analyze not only the execution of the program itself, but also the source code that generated the program. Specifically, I will propose several novel methods that ease the process of understanding program structure and behavior from the software level, and show that these methods subsumes some of the existing phase detection algorithms. My approach is a mix between the top-down approaches that only consider the program execution, and the bottom-up approaches that analyze only the source code.

## 1.1 Contributions

There are three contributions of this thesis. The first contribution is a new methodology that presents an organizational view of the functions and identifies the functions that are responsible for the main steps in the program. The algorithm in the methodology is called LogiView, which works by examining the record of function calls that are made during the execution and selecting several functions as *phase-markers*. These are functions that mark the high-level structures in the program. These functions are usually the main entry points to different components in the program. Any functions that are called exclusively from a phase-marker are considered

to be in the same component and have related tasks.

The LogiView methodology leverages the well-known concept of dominance and dominator trees to provide a novel methodology for organizing program execution and understanding program structure.

The features of the LogiView methodology are:

1. Its input is a record of the function calls that are made during the execution. Specifically, for any pair of functions $A$ and $B$ (not necessarily distinct), the algorithm monitors how often $A$ calls $B$ and under what context.

2. Its output is a set of functions that are selected as phase-markers. These serve to organize a logical view of the various tasks in a program. The tasks are organized hierarchically using a dominator tree. By examining the phase-markers and the dominator tree, one can easily infer the structure of the program. If the program is complex enough, there may be multiple levels of structure. Within a phase, there may be subphases which are divided even further. The phase-markers in the dominator tree thus hierarchically captures the program structure.

Secondly, I use the LogiView methodology to demonstrate that functional redundancies between different programs can be detected. This result helps to identify the objective of a program. For example, if program $A$ is known to solve a linear optimization problem and one detects a functional redundancy between program $A$ and program $B$, then program $B$'s objective is also solving a linear optimization problem or something similar.

This work assumes that programmers don't name their functions randomly. When a program with well-chosen function names is executed, the record of function calls provides semantic information about the program. Furthermore, the technique also leverages the order in which these functions appear in the dominator tree to extract additional relevant information.

Finally, I also show that the LogiView technique can be used in the context of the phase detection problem. I demonstrate that the entries and exits from phase-markers tend to coincide with significant changes to observable physical characteristics. Thus,

if a phase detection algorithm observes the same program execution, it should detect phase boundaries at or near the entries and exits from phase-markers. Phase change can be understood as a point where the behavior changes significantly from the behavior in the past. For example, a program may access data that are not stored in the cache (and thus suffer from a burst of cache miss). We will see that the phase changes in the program can be explained in terms of phase-markers and function entries and exits. Out of 9 phase-markers that we recognize, 8 indeed exhibit phase changes (89% accuracy rate), and out of 11 functions that correspond phase changes, 8 are recognized as phase-markers (73% recognition rate).

I will also show that the program behaviors change at the entries and exits of not only the highest-level phase-markers, but also the lower level ones. That is, I leverage the hierarchical dominator tree to capture multiple levels of phases in the observable program characteristics.

## 1.2 Thesis Roadmap

The remainder of the thesis is organized as follows. First, prior work in understanding program structure and behavior is examined in Chapter 2. The LogiView methodology to generate the dominator tree and select phase-markers is described in Chapter 3. The effectiveness of LogiView is described in Chapter 4. Chapter 5 presents the technique to leverage this methodology along with function names to detect functional redundancy. Chapter 6 examines the correlation of this technique and phase detection problem. My findings and results are summarized in Chapter 7.

# Chapter 2

# Background and Related Work

In the past, there have been several attempts to automate the process of understanding program structure and behavior. These attempts fall into two main categories: software oriented and hardware oriented. Software oriented approach only analyzes the source code of the program and attempts to understand its structure, while hardware oriented approach monitors the observable physical characteristics throughout the program execution and infers the phase boundaries. My thesis is a combination between these two approaches; it monitors the program execution and leverages the information from the source code to produce an logical view of the program structure.

## 2.1 Software Oriented Approaches

The program source code provides tremendous amount of information about program structure and behavior. Sherwood et al. [34] assert that the physical characteristics of a program directly depend on the way the program traverses its source code. Although the source code is sometimes complex and unwieldy, extracting various information from the source code can provide further insights in understanding its structure and behavior.

Larus [19] proposed a method to find phases in the program. He separated the program into *basic blocks* (a sequence of instructions without any branching instructions or branching targets in the middle), and as the program executes, it moves

from one basic block to another. Thus, the program forms a *trace*, a sequence of basic blocks that are executed by the program. Larus then used Sequitur [29] which parses the the trace into context-free grammar. The top level tokens in the grammar corresponds to the high-level phases in the program. This approach is similar to mine in the sense that this approach concerns the transition from one chunk of code into another. But monitoring the transition at a basic block level is too fine-grained. The program may make branching decisions that are not related to its structure. For example, suppose a variable $x$ is treated differently when it is positive, zero, or negative. Larus' approach would not be able to recognize that structurally these steps structurally similar, because the execution falls into three different basic blocks.

Srinivas and Srinivasan [40] introduced an algorithm to find the *hot spots* in the program, a piece of code that are often executed and is responsible for the majority of program execution. This algorithm also relies on the transition between functions and their frequency. In the process of finding hot spots, this algorithm classifies all the functions in the program into *clusters*. A cluster is a set of functions that interact with each other frequently. Loosely speaking, these clusters correspond to the high-level components in the program. Thus, this work introduced the idea that in the record of function calls and their frequency, there are some information about the relationship and connectedness between functions. My LogiView algorithm to find phase-markers in the program is built upon this concept; it relies on the function call records and their frequencies.

Jakobac et. al [14] proposed an algorithm to classify the classes in a Java system into *regions*. They defined clusters as a group of system elemens that are closely related, or independent of other parts of the system. Their algorithm classified each class as *Processing, Communication*, or *Data* based on the variable and function names that are found in the class. For example, when a function in a class has the word `socket` in it, the class is classified as a Communication class. They used this classification results, along with other records such as dependency diagram and function calls, to separate the classes into regions. My methodology of detecting functionality redundance is closely related to this approach in the sense that the names

that the programmers give in the source code may provide semantic information about the classes. When these semantic information is coupled with the function call records, one can infer some further information about program structure and behavior.

Haran et al. [11] used machine learning techniques to predict whether a program will terminate in failure (due to crash or wrong output value) or succeed, given the number of times each function is called. For example, suppose in the previous executions where function $A$ is heavily executed, the program always crashed. Then either function $A$ causes the program to crash, or it is a routine that the program takes before crashing. Either way, function $A$ is highly correlated with the program failure. This result indicates that there is a relationship between the function count and the program behavior. This idea could be expanded to say that, the frequency of each function's provides valuable information about the program's functionality.

Basit et al. [1] proposed a method to recognize two modules with similar functionality (*clones*) in a big software system. The method relies on the similarity between the sources in order to find clones. For example, if an adversary copies function $A$ into function $B$ and simply renames the variable names, this method would recognize that $A$ and $B$ are clones with ease. Even if the adversary changes the order of several lines without affecting the output, this method would still be able to recognize the clones. However, if the adversary rewrites the whole function, or writes an implementation of another algorithm that achieves the same result, the method would not be able to recognize the clones. My methodology supplements this result by providing a way to recognize if two programs have the same functionality even if they use different algorithms.

The work by Zhang et al. [43] used program trace (sequence of instructions that are executed) to find two programs that *match* (have very similar behavior). If two programs are very similar (perhaps they differ by only a few lines in the source code), the traces of their executions should also be very similar. This result is similar to that of Basit et al. [1] above. The difference is that the algorithm of Zhang et al. observes the physical characteristics of the program execution (the trace), while Basit et al. tries to find similarities in the source. The execution trace is, to some extent, similar

to the function call records in a different granularity level. Thus, my methdology uses the approach in between that of Basit et al. and Zhang et al. My methodology uses both the function call records and the function names in order to determine that two programs have the same functionality.

The closest prior work to mine is by Lau et al. [20]. They monitored the entries to and exits from functions and loops, and they counted how many times each transition happens (the *cost* of each transition). By analyzing the relationships between the costs, their algorithm selects several points in the program that serve as phase boundaries (similar to my phase-markers). The difference between this work and my work is that my work focuses only on transition between functions and not loops. The other difference is that this algorithm only recognizes phase-markers at one level, whereas LogiView produces the dominator tree that provides structural information at multiple levels.

## 2.2 Hardware Oriented Approaches

As the program run, there are several physical characteristics that can be extracted from its execution. For example, Rabbah et al. [30] introduced five metrics that can characterize a program's execution. *Predominant data type* indicates the type of data that are mostly used in the program. *Temporal locality* measures how often data or instructions that have been accessed are reacessed again within a short period of time, *Spatial locality* measures colocation of data addresses in memory. *Parallelism* characterizes the program's ability to be executed concurrently. There are other metrics that one can measure from the program's execution, such as *instruction working sets*, the set of instructions that are used in the last $N$ cycles, and *cache miss rates* that measure the frequency of data accesses that are not in the cache.

These characteristics provide some information about program structure and behavior. By monitoring these characteristics, one can isolate several intervals where the characteristics are very different accross intervals but they stay the same within the interval. These intervals are good candidates of program phases.

There have been several works in the past that monitor physical characteristics during program's execution and extract regularities and recurrences in the characteristics. Dhodapkar and Smith [5, 6] use the record of instruction working sets to detect phases in the program. They asserted that the substantial change in the working sets indicates that the program is executing a different code, and hence the program is in a different phase.

Shen et al. [33] measure temporal and spatial locality in the program and argues that they can be used to detect phases in the program. However, their definition of phases was an interval where the locality is relatively similar. In other words, they used the locality to detect phases in locality. This approach may sound self-fulfilling because the resulting phases may not generalize to other characteristics as well. However, they also proposed a method to predict future phases given the past record of locality. The idea that future phases are predictable suggests that program phases are caused by something more fundamental than the physical characteristics. In other words, program phases are not derived by its physical characteristics, but rather, physical characteristics help us to uncover phases.

Similarly, Zhong et al. [44] predict phase changes by measuring data reuse distance. *Reuse distance* of a data $D$ is the number of different data that are accessed between two consecutive accesses to $D$. Although it is a theoretical and abstract metric, reuse distance has a correlation to cache miss rates. Intuitively, if this distance is greater than the cache size, then the second access to $D$ would be a cache miss. But if the distance is less than the cache size, then the second access would be a cache hit, because $D$ would still be in the cache upon the second access.

Zhong et al. also introduce the idea of hierarchical phase. There may not be only one level of phase in the program. At the highest-level, a program may be composed of several phases, but within each of these phases, there may exist subphases. These subphases may exhibit different behaviors just like different phases behave differently. But the difference accross subphases in the same phase is small compared to the difference accross different phases. This idea goes in line with the fact that the program structure is hierarchical as well. Within a program, there may be several modules, and

each modules contains several classes, and each class has several functions. Although Zhong et al. was not able to correlate the different levels of phases that they detected with the granularity levels in the program, they showed that it is indeed possible to uncover many tiers of phases from physical characteristics. Conversely, one can argue that the physical characteristics in the program contain multiple levels of variations. Thus, any phase detection algorithm that are based on physical characteristics should attempt to detect phases at multiple levels.

All of the hardware-oriented phase detection algorithm [5, 6, 33, 44] rely on some kind of instrumentation tools to help them gather the physical characteristics as the program executes. There has been several improvements in the instrumentation techniques. Mysore et al. [27] provided an algorithm to generate program *profile* (aggregate result of physical characteristics over the program execution) with minimal memory requirements. Although the resulting profile is presented as a range (the lower bound and upper bound of the profile), Mysore et al. proved that the range can be made arbitarily small and accurate.

Metz et al. [24] also proposed a method of sampling instrumentation. Instead of monitoring the program execution every instruction, the sampling instrumentation only take measurements every once in a while, in the places that have been precomputed. They proved that their sampling method produced a profile that is quite close to the actual profile (to produce an exact profile is proven to be NP-hard, and they gave an approximation algorithm that produces a profile within 50 percent of the desired profile).

With these sophisticated instrumentation techniques, more improvements are possible to the existing hardward-based phase detection algorithms. Nevertheless, a further advance in this field is difficult as long as we only observe the external characteristics. The physical characteristics and program structure are caused by its source code and not vice versa, [34] thus my work attempts to understand program structure and behavior from a software point of view.

# Chapter 3

# LogiView Algorithm

## 3.1 Overview

The LogiView Algorithm is designed to identify several functions as *phase-markers*, and to produce a dominator tree that presents an organizational view of the functions in the program. The phase-markers are intended to mark the high level structures in the program. Given the dominator tree and the phase-markers, we can easily separate the program into different components by grouping the functions into logival views of the program, based on the dominance of the phase-markers.

This rest of this chapter is laid out as follows. First we will describe the necessary terminologies and theorems to construct a dominator tree. Then we will give an example of a dominator tree and walk-through an investigation of a subtree in it. We will describe several undesirable properties of a dominator tree and show some heuristics to counter these properties. Finally we will introduce a novel algorithm to select the phase-markers based on the structure of the tree.

## 3.2 Call Graph

When an application is executed, we build a *Call Graph* of the execution. The call graph is a directed graph $(V, E)$ where each node $X \in V$ corresponds to one function $f_X$ in the application. There is an edge $(A, B) \in E$ if and only if function $f_A$ calls

function $f_B$. Each edge has a *count* field that shows how many times the call from $f_A$ to $f_B$ occurs.

There is a *start node* that corresponds to the first function that is executed. Typically this function is the `main` method.

If a function $f_Y$ is never called over the execution of the application, then the node $Y$ is omitted from the graph. Therefore, our call graph is always connected because every node $X$ corresponds to a function that has been called. The stack trace during $f_X$'s execution shows a path from the start node to $X$. Thus, for each node $X$, there is at least one path from the start node to $X$.

In order to build the call-graph, we instrument program binaries with PIN (Program Instrumentation Tool) [21]. The instrumentation allows us to intercept entries and exits to each function as the program executes. By keeping track of the functions that are being executed (i.e. functions that the program has entered but not exited yet) we can infer which function is calling the current function. In other words, we simulate the program call stack at all times, pushing and popping functions as signaled by the instrumentation tool.

The call-graph looks like the call trace generated by gprof, a well-known GNU Profiling Tool [10]. Gprof is a program profiling tool that records the function calls during the program execution and summarizes them in a table. Gprof also features exporting this table to a directed graph. The resulting graph is the same graph as our call-graph.

## 3.3 Dominator Tree

We say that the node $A$ *dominates* $B$ if and only if every path from the start node to $B$ must pass through $A$. In other words, every time $f_B$ is executed, its stack trace contains $f_A$. We also say that $A$ is a *dominator* of $B$.

By definition, every node dominates itself. If $A$ dominates $B$ and $A \neq B$, we say that $A$ *strictly dominates* $B$. From the definition of dominator, we can prove the following properties (for a reference to existing proofs, see [26]):

**procedure** $FindFominators(N)$

1: $n_0$ is the start node
2: $Dom(n_0) = \{n_0\}$
3: **for all** $n \in N - \{n_0\}$ **do**
4: $\quad Dom(n) = N$
5: **end for**
6: **while** there are changes in any $Dom(n)$ **do**
7: $\quad$ **for all** $n \in N - \{n_0\}$ **do**
8: $\quad\quad Dom(n) = \{n\} \bigcup \left( \bigcap_{p \in pred(n)} Dom(p) \right)$
9: $\quad$ **end for**
10: **end while**

Figure 3-1: An algorithm that outputs the dominators of each node [26] pp. 182.

1. If two distinct nodes $A$ and $B$ both dominate $C$, then either $A$ strictly dominates $B$ or $B$ strictly dominates $A$.

2. For each node $X$, there is a node $P_X \neq X$ such that $P_X$ dominates $X$ and each node that dominates $X$ also dominates $P_X$. We call $P_X$ an *immediate dominator* of $X$. Each node has one and only one immediate dominator, except the start node, which has none.

Therefore, one can construct a tree of immediate domination, where each node is the children of its immediate dominator. We call it the *dominator tree*. The nodes in the tree correspond to the nodes in the call graph. We say that node $X$ in the dominator tree corresponds to node $C_X$ in the call graph. The root of the dominator tree corresponds to the start node, and $Q$ is a child of $P$ if and only if $C_P$ is an immediate dominator of $C_Q$.

The *depth* of a node is its distance from the start node in the tree. Thus, the start node has depth 0, and the nodes that are directly dominated by the start node has distance 1, and so on.

We build the dominator tree with the algorithm from [26] pp. 182, as shown in Figure 3-1. This algorithm computes the dominators for each node. To build the tree, we realize that the number of dominators of a node equals its depth. Thus, to find the immediate dominator of $X$, we look for a node in $X$'s dominator list whose depth is one less that $X$.

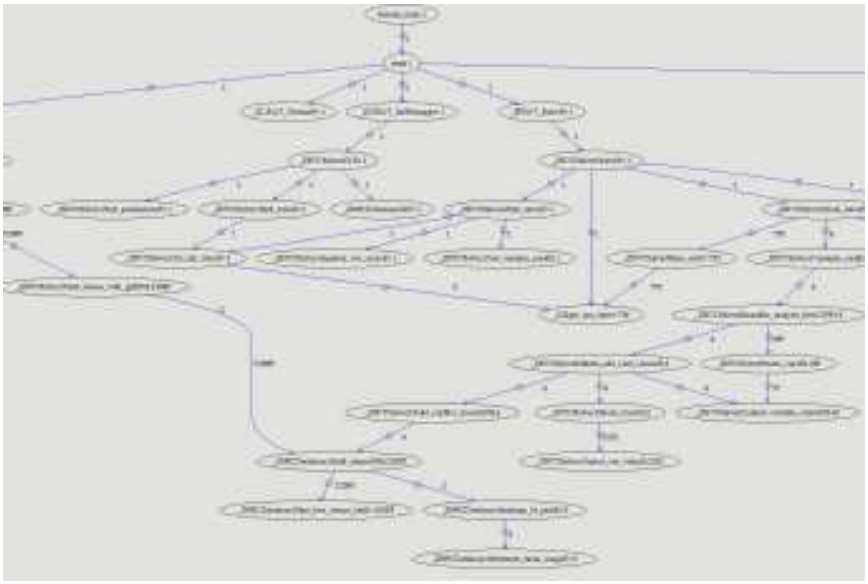

Figure 3-2: A section of the call graph from zChaff benchmark.

The nodes in the dominator tree are also annotated by the number of times the corresponding function is called. We call this the *cost* of the node. We easily obtain the cost of node $X$ from the sum of the count of incoming edges to $C_X$.

We run our algorithm on the zChaff benchmark [9]. It is a benchmark that solves the satisfiability problem by trying to find a satisfying assignment to a boolean formula in Conjunctive Normal Form. Figure 3-2 shows a section of the call graph that is formed during the program execution, and Figure 3-3 shows the corresponding dominator tree. We see that the call graph is very complex and unwieldy. It is hard to understand the relationships between functions in the graph. On the other hand, the dominator tree offers an alternative perspective of the call graph, where the functions are arranged in a hierarchical view. As we shall see, an inspection to this dominator tree could provide us with some insights about the program structure.

Figure 3-5 shows the pseudocode of the zChaff benchmark [9]. Figure 3-4 is a section of the dominator tree in Figure 3-3, showing only the subtree that are dominated in `analyze_conflicts`. Because `analyze_conflicts` dominates all these nodes, thus the corresponding functions are only called directly or indirectly by `analyze_conflicts`. Thus, their tasks are most likely related to the task of conflict analysis.

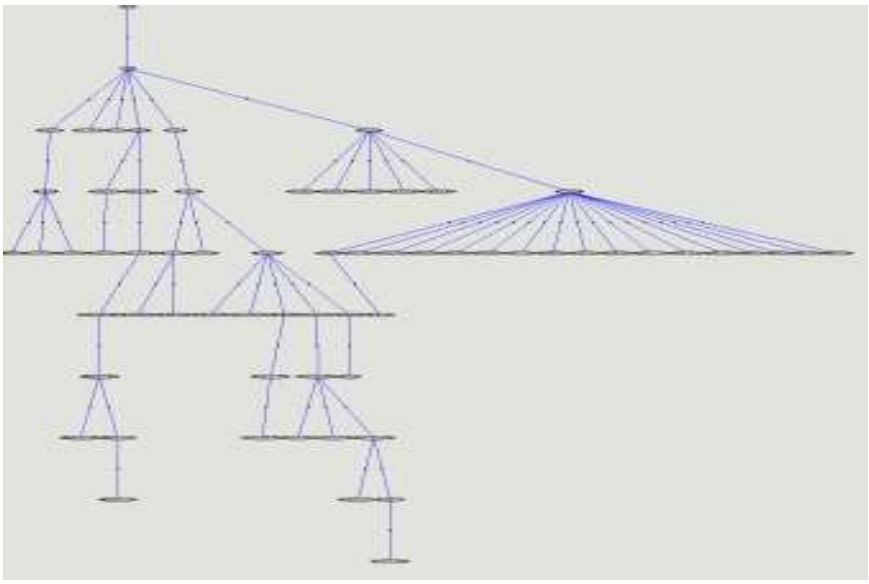

Figure 3-3: The dominator tree from zChaff benchmark.

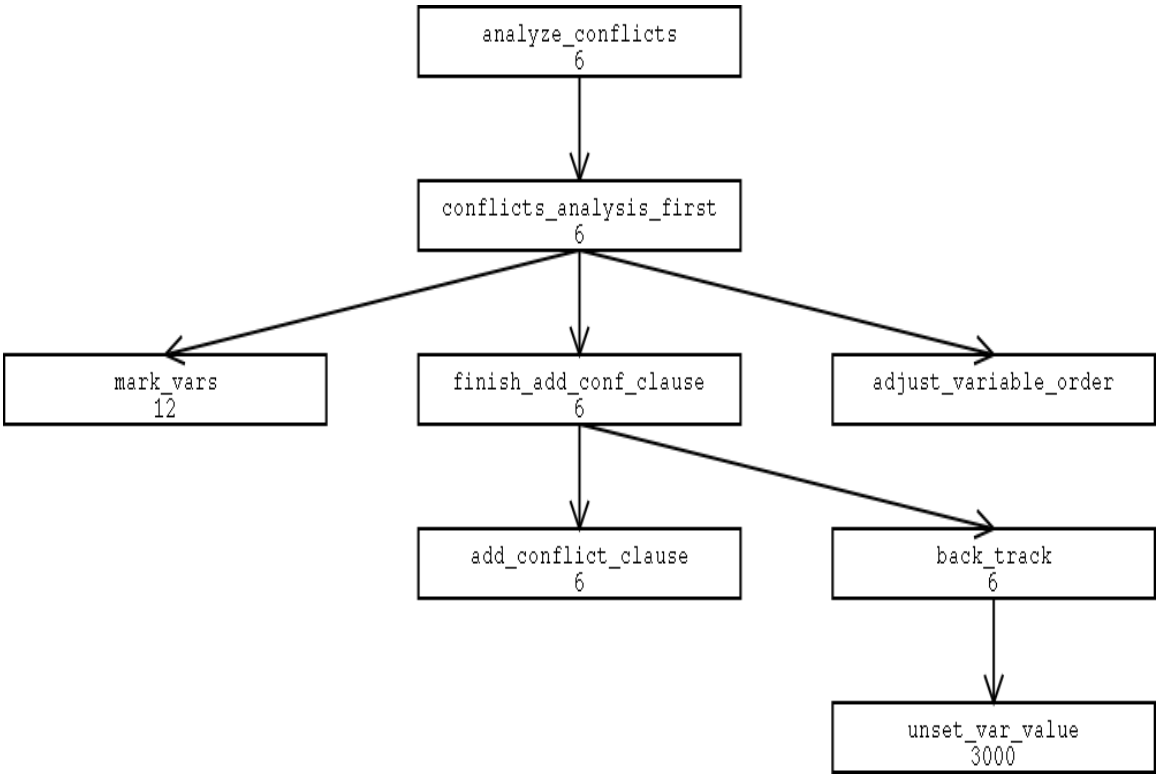


Figure 3-4: A section of the dominator tree from zChaff benchmark.

```
procedure SAT()
 1: while true do
 2:   Decide next branch
 3:   while true do
 4:     status = deduce();
 5:     if status = CONFLICT then
 6:       blevel = analyze conflict()
 7:       if blevel = 0 then
 8:         return Unsatisfiable
 9:       else
10:         backtrack(blevel)
11:       end if
12:     else if status = SATISFIABLE then
13:       return Satisfiable
14:     else
15:       break
16:     end if
17:   end while
18: end while
```

Figure 3-5: The pseudocode of the zChaff benchmark.

Figure 3-5 shows the pseudocode of the zChaff benchmark [9]. From the names of the functions, we can infer that these functions include line 6 and 10 in Figure 3-5. We can even speculate that the subtree in Figure 3-4 corresponds to the loop in line 6-11 of Figure 3-5, because the other lines in that interval are trivial instructions.

In the example above, we see that the function `analyze_conflicts` dominates several functions whose task is related to conflicts analysis. Thus, `analyze_conflicts` is a candidate for a phase-marker, because it is a main entry to a component that is responsible for conflicts analysis. All functions in this component are called from `anaylze_conflicts`, directly or indirectly.

We see that the dominator tree provides an organizational view of the functions in the program. By examining a subtree and the functions it contains, we can infer some information about its task and characteristics. Moreover, if we know the algorithm that is used in the program, we can even correlate some subtrees to specific lines or loops in the algorithm. The dominator tree provides us with a hierarchy under which the functions are grouped according to their tasks. The LogiView algorithm helps the

investigation of the dominator tree by selecting several functions as phase-markers. Loosely speaking, phase-markers are the functions are likely to have subtrees that provide us with the most information about program structure.

## 3.4 Shared Node

A shared node is a node that is called by two different functions in the program. A shared node has multiple paths from the start node, and its dominators are the common nodes in those paths. Very often, a shared node is immediately dominated by the start node. Thus, the shared node is located high in the tree, not because the function is within a few calls from `main`, but because there are two very different paths from `main` to this function. This is problematic, because an ideal phase-marker is also located high in the tree because the phase-marker is likely to be within a few calls from `main`.

We observe that in most programs, there are two basic types of shared nodes. First, a shared node corresponds to a function that is a computational primitive, and thus is called from various places in the program. Library functions, I/O functions, or mathematical routines are such examples. These nodes tend to be called from very different places in program and as a result, they generally have low depth (they are located near the top of the tree). But these nodes are easy to recognize because they are so specialized that they usually don't call other functions, or if they do, they call only a few other functions. Thus, these shared nodes usually have a small subtree dominated by them, or they appear as leaves.

Second, a shared node corresponds to a function that is designed to do a specific task in a particular context, but once in a while it is called to perform the task in a different context. An example involving a shared node of the second kind is depicted in the code in Figure 3-6, taken from the simplified snippet of the zChaff benchmark.

Figure 3-7 shows a section of the call graph and dominator tree that would result from the code in Figure 3-6. In this scenario, `analyze_conflicts` is dominated directly by `solve`, along with `init_solve` and `deduce`. However, from the pseudocode

```
solve() {
  init_solve();
  for( i = 1 to 500) {
    deduce();
  }
}

init_solve() {
  ...
  analyze_conflicts();
}

deduce() {
  ...
  analyze_conflicts();
}
```

Figure 3-6: A sample program that would cause a shared node of the second kind. `analyze_conflicts` is shared between `init_solve` and `deduce`.

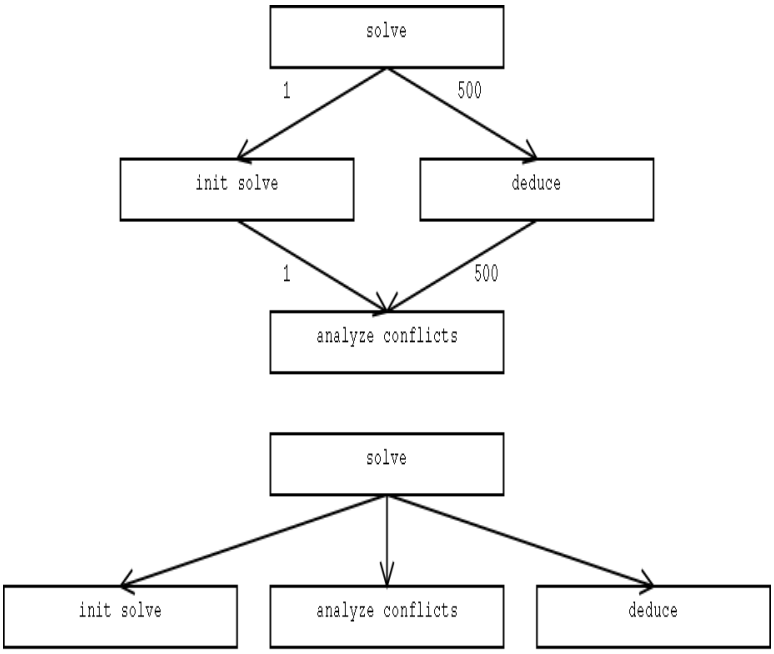


Figure 3-7: A section of the call graph and dominator tree that would result from Figure 3-6.

in Figure 3-5, it seems that `analyze_conflicts` should instead be dominated by `deduce`, because the task of conflicts analysis are a logical "follow-up" of the deduction process. After one variable is assigned a truth value in `deduce`, `analyze_conflicts` ensures that the rest of the assignments are consistent and partially satisfy the boolean formula. Here we see that `analyze_conflicts` is being called under two different contexts. The first context, when it is called by `init_solve`, is to check the assignment for conflicts *before* the loop begins. The second context, when it is called by `deduce`, is to check the assignment for conflicts after one more variable has been assigned a truth value. This second context is indeed the *steady state* context, and is more interesting in terms of understanding program structure than the first context. Indeed, when we examine the pseudocode in Figure 3-5, we see that the conflicts analysis routine is structurally a part of the deduction process.

We see that the presence of a shared node of the second kind could eventually "promote" nodes and subtrees up in the tree. As in the zChaff example, we could no longer infer that conflicts analysis routine is structurally a part of the deduction process because `analyze_conflicts` (and everything it dominates) are promoted up. In order to minimize the effect of a shared node of the second kind, we introduce an edge-pruning algorithm. When a function is called mostly from one place and rarely from another place, we prune (delete) the edge that is less frequent.

We prune the edges in the call graph by looking at each node's incoming edges. For node $X$, let $m_1, \ldots, m_k$ be the count of the incoming edges to $X$, and let $M = \max\{m_1, \ldots, m_k\}$. We delete the incoming edges with count less than $M/C$, where $C$ is a parameter. The value of $C$ should always be greater than 1. A higher $C$ value would result in a more permissive pruning algorithm where fewer edges are deleted and more shared nodes are allowed. A lower $C$ value would result in a stricter pruning algorithm where very few shared nodes are allowed. We found that $C = 2$ works reasonably well for most programs.

```
X::get_data(args) {
  this.y.get_data(args);
}
Y::get_data(args) {
  this.z.get_data(args);
}
Z::get_data(args) {
  return data;
}
```

Figure 3-8: A delegation of task that causes a chain pattern in the dominator tree.

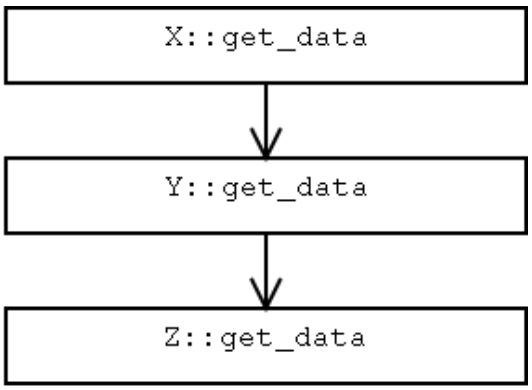


Figure 3-9: The dominator tree formed by the code in Figure 3-8.

## 3.5 Pathological Patterns

In order to find the phase-markers in the program, we have to avoid chain patterns and leafy patterns.

### 3.5.1 Chain Pattern

Chain pattern is a series of nodes where one node is dominated by another in a sequential manner. For example, $(A_1, A_2, \cdots, A_n)$ is a chain-pattern if $A_k$ immediately dominates $A_{k+1}$ and only $A_{k+1}, k = 1, 2, \ldots n-1$.

The chain pattern usually happens from task delegation between objects. The code is Figure 3-8 would cause a chain pattern of length 3, as shown in Figure 3-9. As the task delegation is propagated accross more classes, the chain gets longer, and the topmost node in the chain will seemingly have a large subtree that it dominates.

In other words, we see a large number of functions whose task are seemingly related, because they are all in the same subtree. In this example, all the three functions appear to have a related task of retrieving the data. However, we should not consider the function `X::get_data` a phase-marker, because it does not dominate a component in the program. Structurally, the entire subtree does one task of retrieving the data, not three related tasks. In other words, the subtree could be replaced by one function that retrieves the data (although this function may break the program abstraction). Thus, although chain pattern is an indication of good software engineering principles, the top node of the chain is not an ideal candidate of a phase-marker.

### 3.5.2 Leafy Pattern

Another pattern that we want to avoid is the leafy pattern. This pattern is where the all the nodes in the subtree are immediately dominated by the root of the subtree $R$. In other words, all nodes in the subtree except one are leaves. The dominator tree in Figure 3-7 is an example of leafy pattern of size 3.

The leafy pattern indicates a lack of organization in the code. When node $A$ is immediately dominated by the root, it is either because $A$ is called directly by the root, or $A$ is called by two functions that has no other common dominators than $R$. Moreover, if $A$ is a leaf, then generally $A$'s task is simple enough that it doesn't dominate another node. Thus, a subtree of with leafy pattern is caused by a function $R$ that separates its task into other simple tasks and delegates them to other functions. In Figure 3-7, if `analyze_conflicts`, `init_solve`, and `deduce` were leaves, then `solve` would not be a good candidate of a phase-markers. It would appear merely as a "wrapper" function that delegates its task onto three different functions that are quite simple themselves.

## 3.6 Finding the Phase-Markers

We observe three main characteristics of a phase-marker. First, as we described above, the subtree dominated by the phase-marker should not be of the chain pattern nor

the leafy pattern.

Second, a phase-marker tends to have a large subtree dominated by it. A phase-marker indicates a distinct phase in the program and it usually calls other functions exclusively. For example, if the function `deduce` is a phase-marker, all the functions that are related to deduction process should be called only from within `deduce`, directly or indirectly, causing these functions to be dominated by `deduce`. If `deduce` truly represents the main entry to a non-trivial component in the program, this component should have multiple functions in it. Consequently, the subtree that `deduce` dominates has multiple functions in it.

A tree of size 3 must be of the chain pattern or a leafy pattern. In fact, 4 is the size of the smallest tree that is not of the chain pattern nor the leafy pattern. Hence, we choose a phase-marker only if the subtree it dominates has at least 4 nodes in it.

Third, the frequency of a phase-marker is significantly bigger than that of its parent. Because a phase-marker encompasses a repeated behavior, the phase-marker is called multiple times by its parent. As in the example above, the frequency of `deduce` is 500 times that of `solve`. During the execution of `solve`, there will a pattern that is repeated 500 times. Each repetition corresponds to one execution of `deduce`. Note that `deduce` may dominate some nodes with a smaller frequencies. This is because in each execution of `deduce`, it may make calls to another function occassionally, but not every time.

Using these three criteria, we introduced the following algorithm.

We classify the nodes in the dominator tree into groups. Node $B$ is in a different group from its immediate dominator $A$ if and only if the frequency of $B$ is at least $K$ times the frequency of $A$. It means that for each call of $A$, in average, there are $K$ calls to $B$. The value of $K$ should always be greater than 1. A higher value of $K$ would result in an algorithm that finds fewer phase-markers. But if $K$ is too high, there may be no phase-markers to be found. We found that the values of $K = 2$ to $K = 5$ produce the same phase-markers for most programs, so we arbitrarily choose $K = 5$ for our results analysis.

We start from the top of the tree (the start node) and go down one depth at a

```
procedure FindPhaseMarkers()
 1: M ← the largest depth in the tree
 2: for i = 1 to M do
 3:    for all n such that n has level i do
 4:       p ← n.immediate_dominator
 5:       if n.frequency > K × p.frequency then
 6:          n.group ← a new group
 7:       else
 8:          n.group ← p.group
 9:       end if
10:    end for
11: end for
12: for all n such that n.group ≠ n.immediate_dominator.group do
13:    if the subtree n dominates satisfies the subtree sanity condition then
14:       print  n is a phase-marker
15:    end if
16: end for
```

Figure 3-10: An algorithm that finds the phase-markers in the dominator tree.

time. If a node has a frequency greater than $K$ times the frequency of its parent, it is allocated to a new group, otherwise it is assigned to the same group as its parent. For each node $n$ with a different group than its parent, we check the following conditions (which we shall call the *subtree sanity conditions*):

1. The subtree $n$ dominates is not of the chain pattern. In other words, $\exists x, y, z$ such that $n$ dominates $x$ and $x$ immediately dominates $y$ and $z$.

2. It is not a leaf. In other words, $\exists x$ such that $n$ dominates $x$

3. The subtree $n$ dominates is not of the leafy pattern. In other words, $\exists x, y$ such that $n$ dominates $x$ and $x$ dominates $y$.

4. The subtree $n$ dominates has at least 4 nodes.

If the subtree that $n$ dominates satisfies the subtree sanity conditions, we mark $n$ as a phase-marker.

The pseudocode for this algorithm is shown in Figure 3-10. Consider a sample dominator tree in Figure 3-11. We will simulate the algorithm through this tree, with $K = 5$.

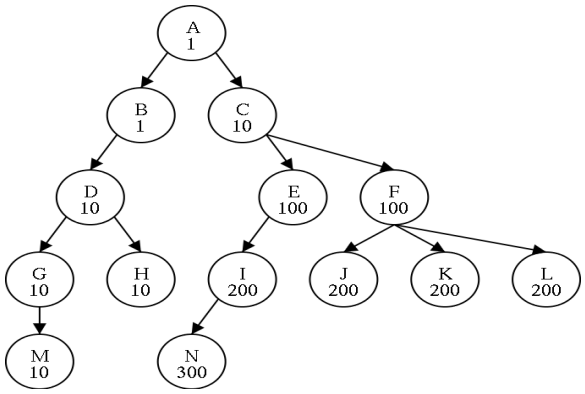


Figure 3-11: A hypothetical dominator tree.

Line 6 will be executed for $n = C, D, E, F$, and line 8 for all other nodes. So, at the end of line 11, the formed groups are $\{A, B\}$, $\{C\}$, $\{D, G, H, M\}$, $\{E, I, N\}$, and $\{F, J, K, L\}$. Then line 12 is executed for the nodes $C, D, E, F$. However, $E$ does not satisfy the subtree sanity conditions because it is of a chain pattern, and $F$ does not satisfy the subtree sanity conditions because it is of a leafy pattern. On the other hand, $C$ and $D$ satisfy the subtree sanity conditions. Thus the phase-markers are $C$ and $D$.

We see that the phase-markers are not necessarily at the top of the dominator tree. $D$ is a phase-marker although it is not at the top. This is because $A$ only calls $B$ a few times, but $B$ calls $D$ many times. One could speculate that the call from $A$ to $B$ may be a delegation call where $A$ delegates its task to $B$. Alternatively, the call from $A$ to $B$ may be an initialization call. Either way, we are not very interested in the call from $A$ to $B$ because it only happens once. We are instead interested in the call from $B$ to $D$ because it happens multiple times, and because $D$ has a substational subtree it dominates.

The phase-markers in this hypothetical tree are $C$ and $D$. Thus we can expect that the subtrees dominated by them to form a logical component in the program. Specifically, we expect that the functions $(D, G, H, M)$ work together to accomplish a certain task, while the functions $(C, E, F, I, J, K, L, N)$ work together for another task. Further, given the algorithm or the pseudocode of this program, we expect that these two sets of functions correspond to significant steps in the pseudocode. In the

next chapter, we run LogiView on a number of benchmarks and demonstrate that the subtrees dominated by phase-markers indeed correspond to the significant steps in the algorithm.

# Chapter 4

# Experiments and Analysis

The LogiView algorithm is designed to identify the functions that mark the high level structures of the program. One way to measure the effectiveness of the algorithm is by comparing the phase-markers against the pseudocode describing the input program. In the ideal case where the phase-markers truly reflect the high level structures of the program, the subtrees that are dominated by the phase-markers correspond to the components that are meaningful in understanding the code organization. The phase-markers would also coincide with the set of functions in the pseudocode.

However, there are some limitations of this approach that make it impractical to define a closed form metric for evaluation.

1. This approach relies that the program is well-designed and the programmer divides the work of the program into functions that are meaningful and well-organized. Some poorly designed programs could contain only one or a few functions for the entire program and they are not organized in any way. Some other programs could be divided into functions irrespective of the pseudocode. That is, one statement / line in the pseudocode corresponds to multiple functions in the program, whereas another block of multiple lines in the pseudocode corresponds to one function in the program

2. Although the programmer may design the program and divides the task into functions according to the pseudocode, the names of the function may be dif-

| Phase-marker | Phase-parent |
|---|---|
| ucxx1 | uloop |
| ucxx2 | uloop |
| uloop | utemp |
| unlap | utemp |

Table 4.1: The phase-markers from the Timberwolf Benchmark and their parents.

ferent from that of the pseudocode. Programmers may choose to name their functions according to their functionality and these names may not resemble anything in the pseudocode. Thus, relying on the pseudocode alone to judge the components in the program is ineffective.

Based on the limitations above, we do not attempt to define a closed form metric and apply it to all programs, nor will we judge the effectiveness of LogiView from the pseudocode alone. Rather, we will examine each program on a case-by-case basis and show that our phase-markers indeed present a view that helps understanding the code organization.

When examining the phase-markers in the program, we also need to examine their immediate dominators, called the *phase-parents*. The phase-parents provide the context under which the phase-markers are called, and will help us in understanding the functionality of the phase-markers.

## 4.1 Experiments and Results

We apply the LogiView methodology to a number of SPEC benchmarks [39]. The following describe my findings.

### 4.1.1 Timberwolf Benchmark

The first benchmark we shall consider is the 300.twolf benchmark [38]. This benchmark simulates the global routing package system. Given the required connections between a group of transistors, it finds the placement and orientation of the transistors that requires the least amount of resources (physical space, power and ground

```
procedure Algorithm Structure(j_0, T_0)
 1: T ← T_0
 2: X ← j_0
 3: while cost(j) has not changed for 3 consecutive repetitions do
 4:    while less than 20 new states per unit generated at each stage do
 5:       j ← generate(X) { generates a new state j incrementally from the previous
          state X by a weighted random selection}
 6:       if accept(c(j), c(X), T) then
 7:          X ← j
 8:       end if
 9:    end while
10:    T ← update(X).
11: end while
procedure accept(c(j), c(i), T)
 1: Δc ← c(j) − c(i)
 2: y = f(Δc, T)
 3: r ← a random number between 0 and 1 with uniform probability
 4: if r < y then
 5:    return TRUE
 6: else
 7:    return FALSE
 8: end if
```

Figure 4-1: The pseudocode of the Timberwolf Benchmark.

```
 1 main() {
 2   utemp();
 3 }
 4
 5 utemp() {
 6   while ‘‘stopping criterion’’ is not finished {
 7     findunlap();
 8     uloop();
 9   }
10 }
11
12 findunlap() {
13   // computes the cost
14   unlap();
15 }
16
17 unlap() {
18   // sorts and places cells
19 }
20
21 uloop() {
22   ucxx1();
23   ucxx2();
24 }
25
26 ucxx1() {
27   acceptt();
28 }
29
30 ucxx2(){
31   acceptt();
32 }
```

Figure 4-2: The simplified view of the Timberwolf code.

```
122 for( ; ; ) {
...
134   uloop();
...
237   if( iteration >= freeze + 3 || stage == 3 ) {
238     /*
239      * WE ARE FINISHED *
240      */
...
279     break;
...
282   }
283 }
```

Figure 4-3: A snippet from `utemp.c` in the Timberwofl Benchmark.

connections, etc). Figure 4-1 shows the pseudocode of this benchmark from [32]

This algorithm belongs to the class of algorithms called *probabilistic hill-climbing algorithms*. An algorithm in this class finds the maximum value of an objective function by starting at one point. At each iteration, the algorithm tries to improve the objective function by moving in the direction that increases the objective value (hence the name hill-climbing). This direction is found by randomizing some parameters in the algorithm, so that for each iteration the direction of the point movement is not deterministic. The function accept in Figure 4-1 determines if the most recent movement indeed increases the objective function. If it does not, then the move is not executed.

Figure 4-4 shows the call graph for this benchmark, and Figure 4-5 shows the corresponding dominator tree. We see that the call graph is very complex and impractical to investigate, while the dominator tree provides a more organized view of the functions. Table 4.1 shows the phase-markers and the phase-parents for this benchmark. Figure 4-2 shows a simplified view of the code, using mostly functions from Table 4.1.

Function `findunlap` computes the cost of a state, according to the comments in ([38] `main.c` line 225-229). This function calls `unlap` that sorts the cells and places them in a theoretical microchip. Function `findunlap` performs every calculation of

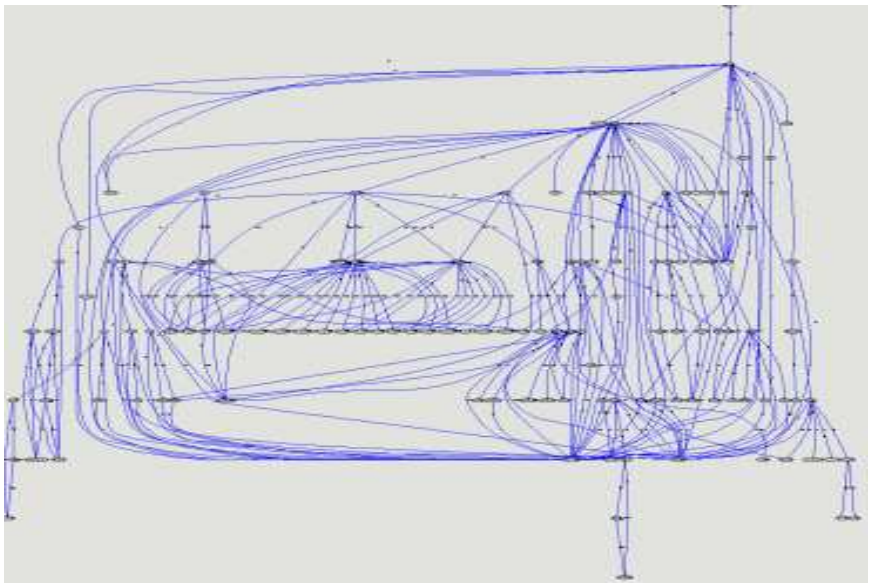

Figure 4-4: The call graph for the Timberwolf Benchmark.

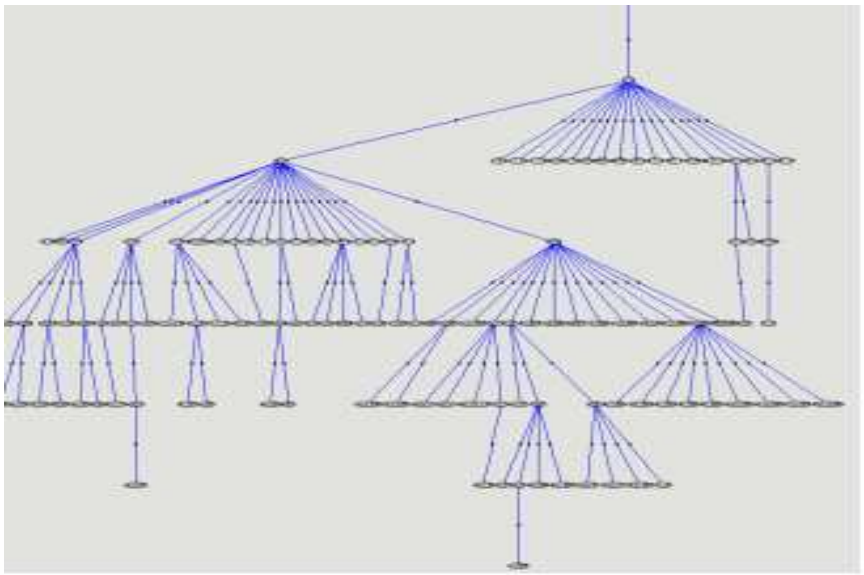

Figure 4-5: The dominator tree of the Timberwolf Benchmark.

$c(X)$ and $c(j)$ in the pseudocode. This function indicates how "expensive" it is to build a physical microchip that is indicated by the state $X$ or $j$. It depends on a number of variables such as the space and power requirements. If one needs to design the algorithm to be biased towards space conservation or power conservation, he only needs to alter the cost function to reflect the bias. In this benchmark, he only needs to examine the function `findunlap` and make the appropriate change.

Figure 4-3 shows a snippet from [38]`utemp.c`. Line 237 corresponds to line 3 in the main algorithm in Figure 4-1. Line 122 causes a loop that will be repeated indefinitely until the condition in Figure 4-1 line 3 is satisfied, because the break statement in line 239 can only be executed if the cost has not changed for 3 consecutive repetitions. Therefore, lines 123-236 in `utemp.c` correspond to lines 4-10 in Figure 4-1. The two functions that are called from this region are `uloop` and `savewolf`. Function `savewolf` is primarily concerned with saving the current state to a file, and corresponds to line 10 in the pseudocode. Thus, function `uloop` corresponds to the loop in Figure 4-1 line 4-9.

The function accept in the pseudocode determines if the new state is worth preserving. It computes the increase in the cost function and tests if the increase is greater than a random number. In other words, it insists that the movement of the point increases the cost function by a large enough value. This function roughly corresponds to the function `acceptt` in the code [38]`acceptt.c`. It takes one parameter `delta_cost`, which suggests that this function corresponds to line 2-8 in Figure 4-1 accept procedure. The first line, the computation of change in cost itself, is done outside of `acceptt`.

Figure 4-6 shows another snippet that demonstrates the relationship between `uloop` and `acceptt`. As the function `uloop` executes lines 4-9 in the pseudocode, it makes calls to `ucxx1` and `ucxx2` depending whether the condition in line 227 is satisfied or not. Both `ucxx1` and `ucxx2` call `acceptt` which executes the function accept in the pseudocode. The change in cost itself is computed in the argument to `acceptt`.

Function `uloop` calls `ucxx1` and `ucxx2` at different points, depending on the char-

```
uloop.c
...
227  if( *cellbptr == 0 ) {
228
229    if( ablckptr->borient == 1 ) {
230      if( bblckptr->borient == 1 ) {
231        if( ucxx1( bxcenter, bycenter)){
...
271  } else { /*   *cellbptr >= 1   */
...
284    if( ablckptr->borient == 1 ) {
285      if( bblckptr->borient == 1 ) {
286        t = ucxx2( ) ;
...

ucxx1.c
...
 65 truth = acceptt(funccost+penalty-cost-newpenal-delta_vert_cost);
...

ucxx2.c
...
111 truth = acceptt(funccost+penalty-cost-newpenal-delta_vert_cost);
...
```

Figure 4-6: A snippet from `uloop.c`, `ucxx1.c`, and `ucxx2.c` in the Timberwofl Benchmark.

| Phase-marker | Phase-parent |
|---|---|
| power_prune | prepare_to_parse |
| print_sentence | batch_process |
| read_entry | read_dictionary |
| restricted_expression | read_entry |

Table 4.2: The phase-markers from the Link Grammar Parser Benchmark and their parents.

```
1. Word expressions are extracted from the dictionary and pruned.
2. Disjuncts are built.
3. A series of pruning operations is carried out.
4. The linkages having the minimal number of null links are counted.
5. A "parse set" of linkages is built.
6. The linkages are post-processed.
```

Figure 4-7: The original pseudocode for the Link Grammar Parser.

acteristics of the state. However, both `ucxx1` and `ucxx2` call `acceptt` ([38] `ucxx1.c` line 64, `ucxx2.c` line 111). which suggests that these two functions correspond to or contain line 10-11 in Figure 4-1.

### 4.1.2 Link Grammar Parser

We will consider the 197.parser benchmark [37]. This benchmark is a syntactic parser of English sentences, based on a link grammar, an original theory of English syntax. Given a sentence, the system parses it and assigns to it a syntactic structure consisting of set of labeled links connecting pairs of words [37].

Figure 4-8 shows the call graph for this benchmark, and Figure 4-9 shows the corresponding dominator tree. We see that the call graph is very complex and impractical to investigate, while the dominator tree provides a more organized view of the functions. Table 4.2 shows the phase-markers and their parents that we found in this benchmark. The original pseudocode for the parser is shown in Figure 4-7 [42]

Figure 4-10 contains the subtree rooted in `read_dictionary` and corresponds to line 1 in Figure 4-7. Both `read_entry` and `restricted_expression` were recognized as phase-markers. From examining this subtree, we can infer that the process of

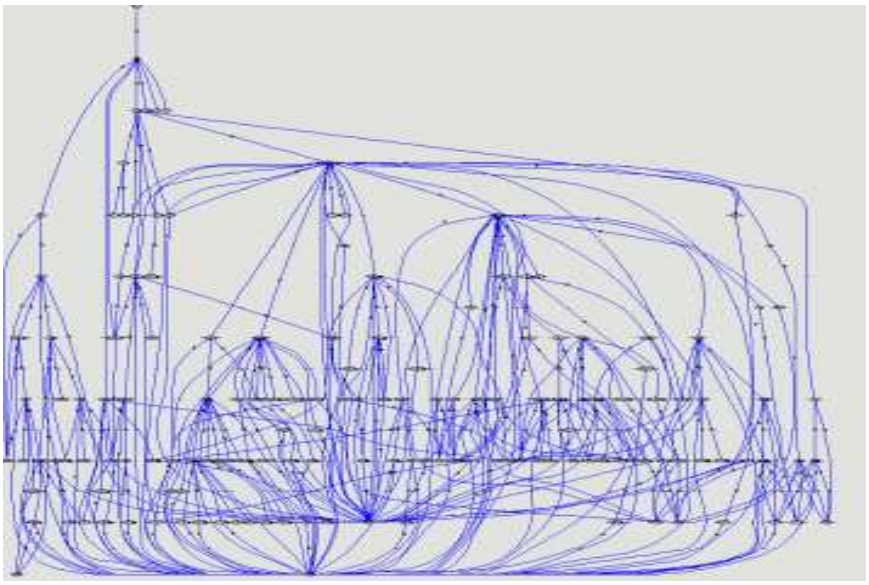

Figure 4-8: The call graph for the Link Grammar Parser.

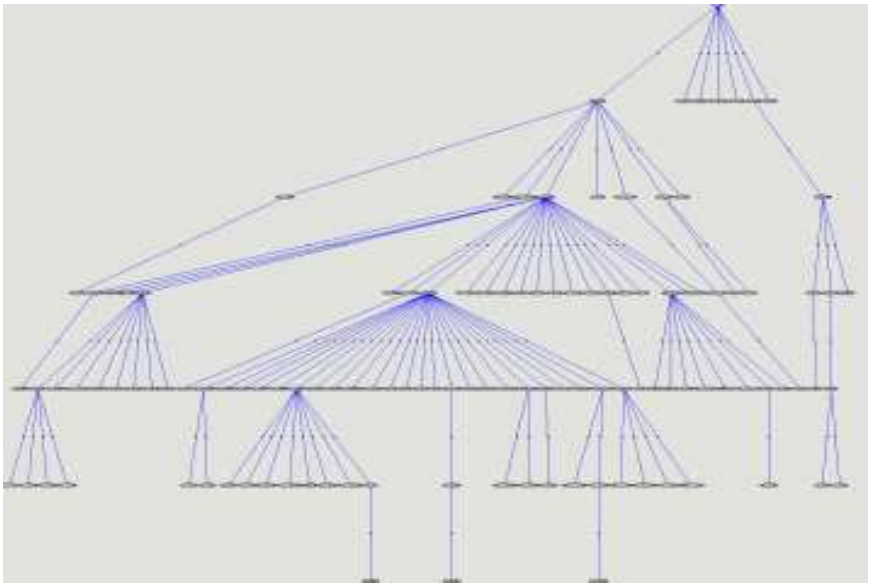

Figure 4-9: The dominator tree of the Link Grammar Parser.

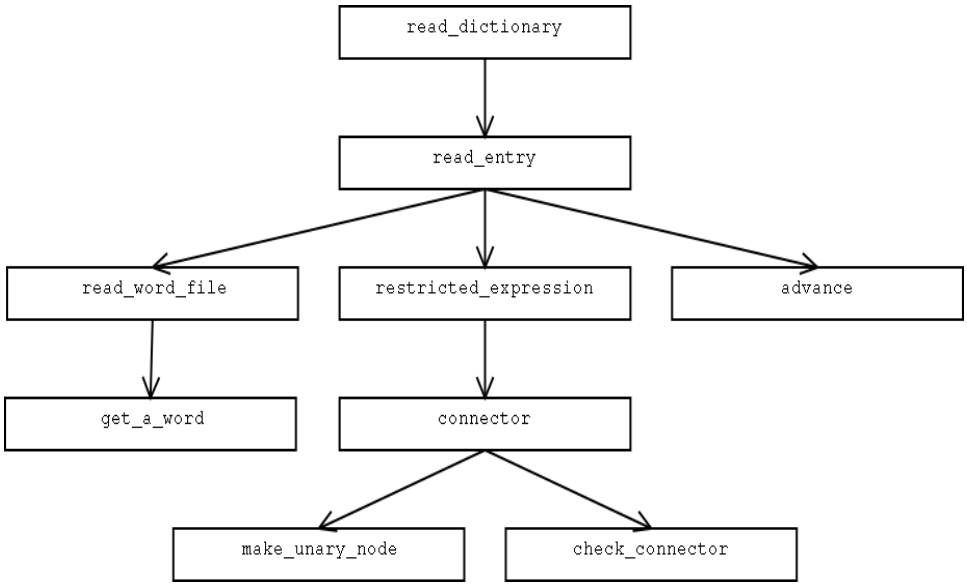


Figure 4-10: The subtree rooted in `read_dictionary` in the SPEC parser benchmark.

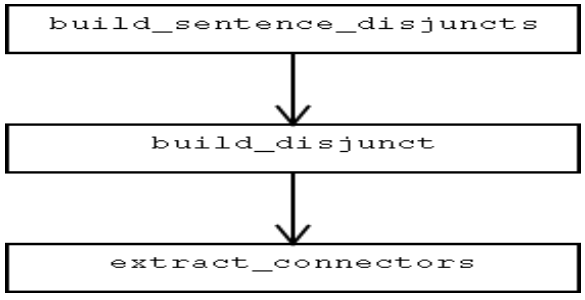


Figure 4-11: The subtree rooted in `build_sentence_disjuncts` in the SPEC parser benchmark.

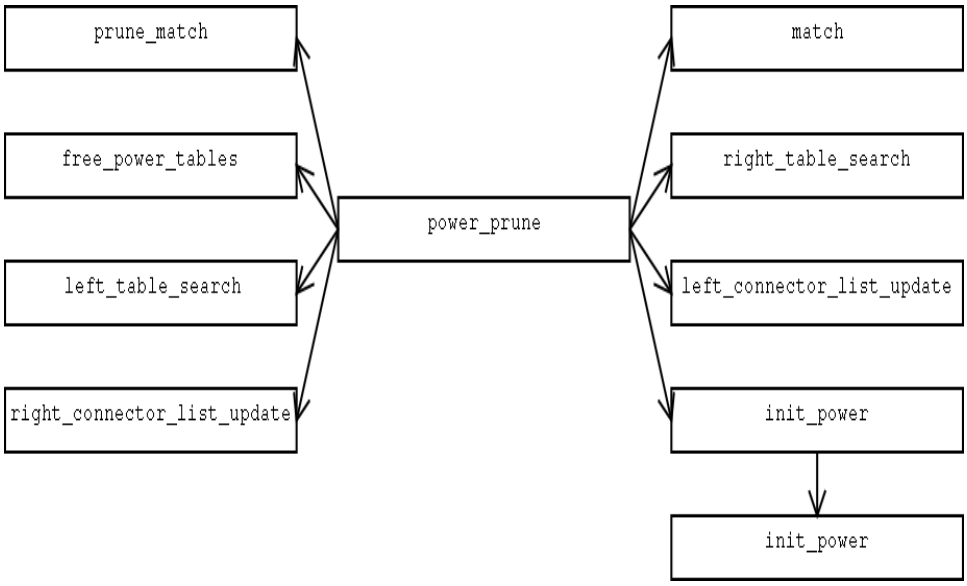


Figure 4-12: The subtree rooted in `power_prune` in the SPEC parser benchmark.

reading the dictionary consists of reading entries, one at a time. The leftmost branch of `read_entry` contains the functions `read_word_file` and `get_a_word`, which suggests that the dictionary entries are represented as word files. The function `advance` is responsible for advancing to the next word file. The middle subtree contains the functions that take the expression from the dictionary and builds the internal representations of the expression.

Similarly, the subtree that corresponds to line 3 in Figure 4-7 is shown in Figure 4-12. The function `power_prune` immediately dominates several functions that are involved in the pruning operation. From the names of these functions, we can speculate that the pruning operation consists of searching for a match between the disjuncts, pruning the match, and updating the various pointers and tables.

The subtree in Figure 4-12 shows the subtree dominated by `power_prune`. The source for function `power_prune` can be found in ([37] `prune.c` line 1282-1364). This function makes a back-and-forth passes along the sentence, and each time it determines if a pruning action is available on the sentence. If so, it performs the required pruning, which involves rearranging the links and connectors between disjuncts.

Figure 4-11 contains the subtree that corresponds to line 2 in Figure 4-7. Our algorithm did not consider `build_sentence_disjuncts` to be a phase-marker because

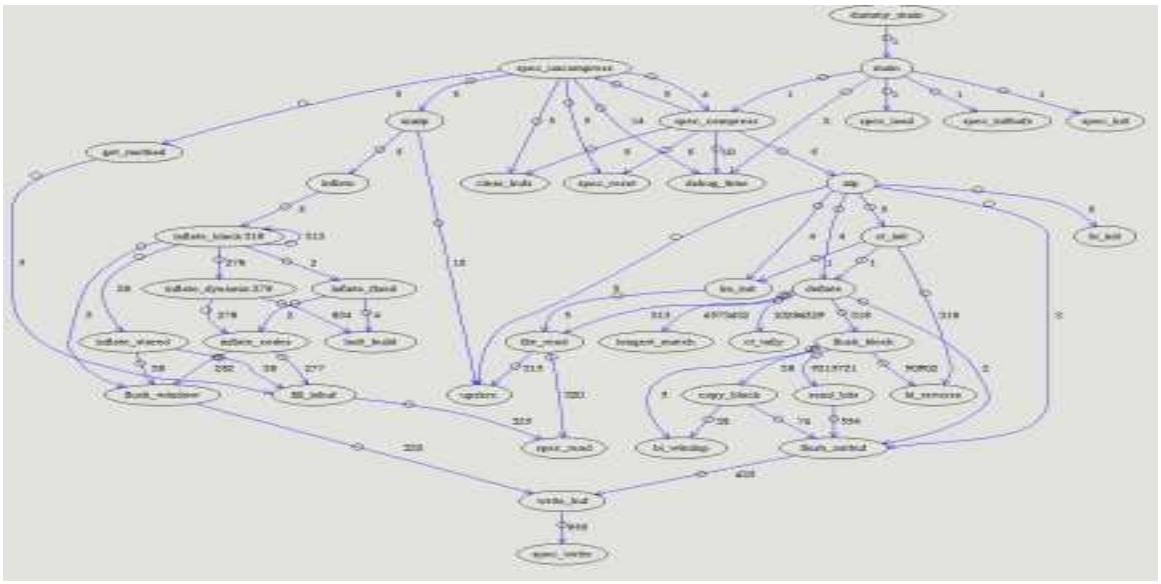

Figure 4-13: The call graph of the GNU Zip Benchmark.

the subtree underneath it is of the chain pattern. The root of this subtree actually satisfied all the other requirements to be a phase-marker, i.e, it is called many more times than its parent, and it contains a node other than itself which is not a leaf. So if we had relaxed the requirements that a phase-marker must not contain a subtree that is of the chain pattern, this subtree would have been flagged as a phase-marker.

The reason why we prevented a chain pattern from being a phase-marker is to prevent long chains of delegations to become a phase-marker. In Figure 4-11 we see an example of a chain pattern. The function `build_sentence_disjuncts` and the function `build_disjunct` are very similar and the call from the former to the latter has some delegative aspects in it. However, `build_disjunct` calls `extract_connectors` and this call is not a delegation. Thus, this subtree exhibits one case where the subtree sanity conditions potentially prevent a good candidate of a phase-marker from being recognized.

### 4.1.3 GNU Zip

We will consider the 164.gzip benchmark [36]. Figure 4-13 shows the call graph and Figure 4-14 shows the corresponding dominator tree. Table 4.3 shows the phase-

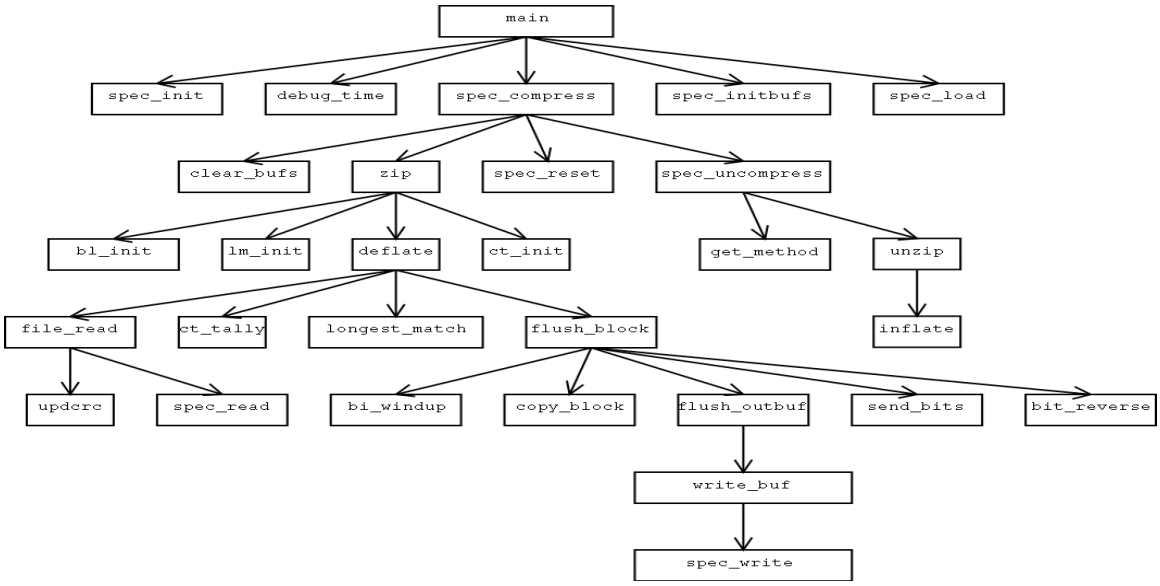


Figure 4-14: The dominator tree of the GNU Zip Benchmark.

| Phase-marker | Phase-parent |
|---|---|
| flush_block | deflate |
| spec_compress | main |

Table 4.3: The phase-markers from the GNU Zip Benchmark and their parents.

markers that were found and their parents. This benchmark performs compression and decompression of a file and hence the pseudocode is very straightforward. Given a file, the benchmark compresses it and then decompresses it. This pseudocode is not interesting to us. But what is interesting is the dominator trees and the phase-markers that are generated by our algorithm. The phase-markers were able to predict the main steps and the substantial aspects of the program execution.

The highest-level phase-marker is `spec_compress`. This function encompasses the actual benchmark to be performed. The nodes outside of the subtree rooted in `spec_compress` are only those functions that are related to SPEC benchmarking routines, such as `debug_time, spec_init`, etc. Thus, we can say that the subtree under `spec_compress` is where the substantial part of the program execution happens.

The function `deflate` is recognized as the parent of a phase-marker. This function is where the actual compression and decompression occurs. Prior to entering `deflate`, the program merely prepares the file and sets up the buffer for reading and writing

files. The compression and decompression algorithm itself happens from within the subtree rooted under `deflate`. Line 7-61 of `deflate.c` [36] describes the compression algorithm using a sliding window and the repeated text of the largest length, and the entire algorithm is dominated under this phase-marker.

Although the pseudocode to GNU Zip benchmark is uninteresting and we only found only two phase-markers, these phase-markers indeed identify the structural phases in the program. The first phase-marker predicts the chunk of the program execution that is unique to GNU Zip benchmark (as opposed to common to all SPEC benchmarks). The parent of the second phase-marker isolates the interval where the actual compression and decompression process occur.

### 4.1.4 Versatile Place and Route

We will consider the 175.vpr benchmark [2].

Versatile Place and Route (VPR) is a placement and routing program that automatically implements a circuit in a Field-Programmable Gate Array chip. There are two major phases of VPR, the placing phase and the routing phase. The placing phase determines which logic block and I/O pad should implement each of the functions in the circuit, while the routing phase determines which programmable switches should be turned on in order to connect the existing wires to the logic block I/O [2]. The two phases are run by invoking VPR twice with different command-line inputs. In this thesis we will examine only the routing phase of the program.

In the routing phase, the connections required by the circuit are represented as a hypergraph and the possible connections are represented by a (different) directed graph. VPR uses a congestion detection algorithm as shown in Figure 4-15 [23]. The heart of this algorithm is actually a variation of Dijkstra's algorithm [2]. The algorithm computes the shortest distance from $s$ to $t$ in a directed graph $G$. Figure 4-16 shows the pseudocode of Dijkstra's algorithm. Lines $1-4$ are only initialization steps, but lines $5-17$ are the actual algorithm itself. These lines compute the shortest distance from $s$ to any node in the graph. Lines $18-24$ are called the *backtracking* steps that actually output the path from $s$ to $t$.

```
procedure NegotiatedCongestion
 1: while shared resources exist do
 2:   for all signals i do
 3:     Rip up routing tree RT_i
 4:     RT_i ← s_i
 5:     while there are new sinks t_ij to be found do
 6:       Initialize priority queue PQ to RT_i at cost 0
 7:       repeat
 8:         Remove lowest cost node m from PQ
 9:         for all fanouts n of node m do
10:           Add n to PQ at cost c_n + P_im
11:         end for
12:       until new t_ij is found
13:       for all n in path t_ij to s_i  do
14:         Update c_n
15:         Add n to RT_i
16:       end for
17:     end while
18:   end for
19: end while
```

Figure 4-15: The Negotiated Congestion algorithm [23].

We see a resemblance of the Dijkstra's Algorithm in the Negotiated Congestion Algorithm. Lines $5-12$ in Figure 4-15 correspond to lines $1-17$ in Figure 4-16 as the main Dijkstra's Algorithm. lines $9-11$ in Figure 4-15 also correspond to lines $11-16$ in Figure 4-16, as they both expand the list of "known distances" from $s$. Line 8 in Figure 4-15 corresponds to line 9 in Figure 4-16. Both lines find the next candidate to be added to the known distance list by extracting the minimum element from a priority queue. Lines $13-16$ in Figure 4-15 are backtracking steps that correspond to the lines $18-24$ in Figure 4-16 [23].

Figure 4-8 shows the call graph for this benchmark, and Figure 4-9 shows the corresponding dominator tree. We see that the call graph is very complex and impractical to investigate, while the dominator tree provides a more organized view of the functions. Table 4.4 shows the phase-markers in the VPR benchmark and their parents.

Figure 4-19 shows the subtree dominated by `route_net`. Figure 4-20 shows the

**procedure** $Dijkstra(G, w, s, t)$

1: **for all** vertex $v \in V(G)$ **do**
2: $v.d \leftarrow \infty$
3: $v.previous \leftarrow$ undefined
4: **end for**
5: $s.d \leftarrow 0$
6: $S \leftarrow \emptyset$
7: $Q \leftarrow V(G)$
8: **while** $Q \neq \emptyset$ **do**
9: $u \leftarrow$ ExtractMin($Q$)
10: $S = S \bigcup \{u\}$
11: **for all** $(u, v) \in E(G)$ outgoing from $u$ **do**
12: **if** $u.d + w_{uv} < v.d$ **then**
13: $v.d \leftarrow u.d + w_{uv}$
14: $v.previous \leftarrow u$
15: **end if**
16: **end for**
17: **end while**
18: $S' \leftarrow \emptyset$
19: $u' \leftarrow t$
20: **while** $u.previous$ is defined **do**
21: Insert $u$ to the beginning of $S'$
22: $u \leftarrow u.previous$
23: **end while**
24: **return** $S'$

Figure 4-16: The Dijkstra's Algorithm to find the shortest path from $s$ to $t$ in a directed graph $G$.

| Phase-marker | Phase-parent |
|---|---|
| build_rr_xchan | alloc_and_load_rr_graph |
| build_rr_ychan | alloc_and_load_rr_graph |
| check_adjacent | check_route |
| expand_neighbours | route_net |
| get_tok | read_net |
| node_to_heap | expand_neigbours |
| route_net | try_route |

Table 4.4: The phase-markers from the VPR Benchmark and their parents.

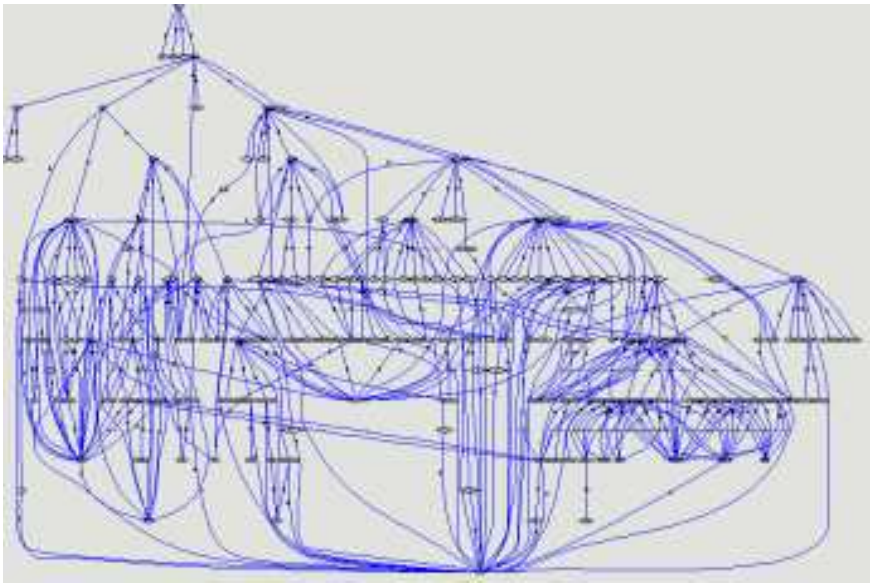

Figure 4-17: The call graph for the VPR.

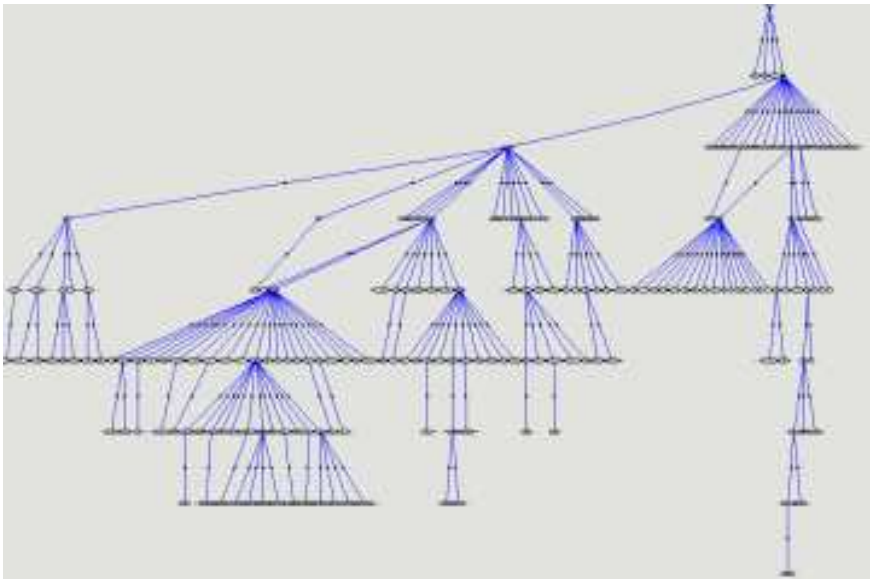

Figure 4-18: The dominator tree of the VPR Benchmark.

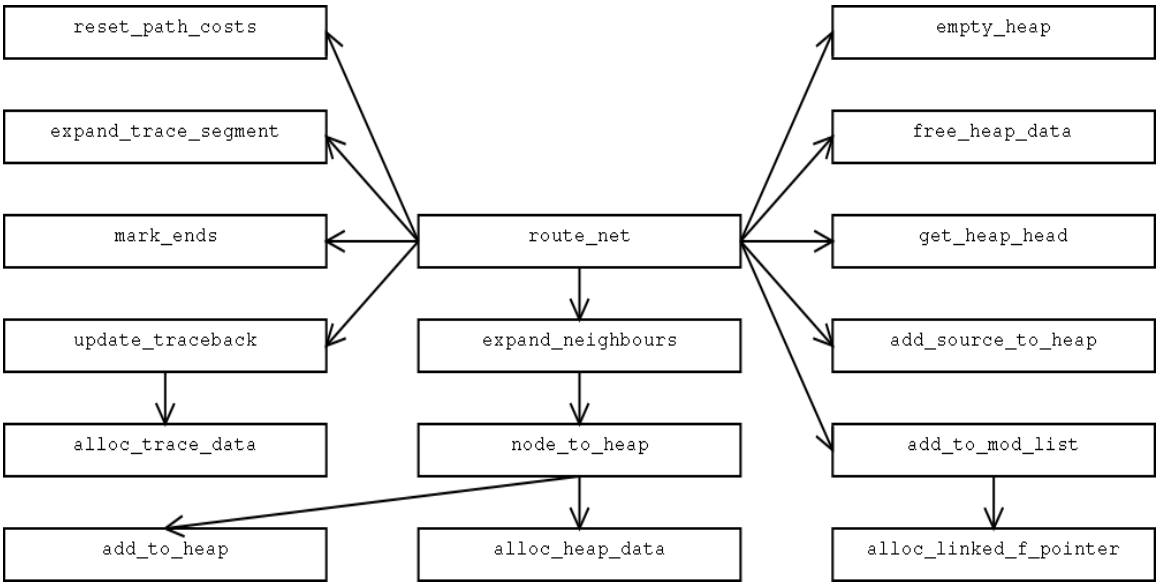


Figure 4-19: The subtree dominated by `route_net` in the VPR benchmark.

```
479 static boolean route_net (...) {
...
507   for (i=1;i<net[inet].num_pins;i++) {
508     expand_trace_segment (tptr, remaining_connections_to_sink);
509     current = get_heap_head();
...
519     while (rr_node_route_inf[inode].target_flag == 0) {
...
522       if (pcost > new_pcost) {      /* New path is lowest cost. */
...
530         expand_neighbours (inode, new_pcost, inet, bend_cost);
531       }
532       free_heap_data (current);
533       current = get_heap_head ();
...
541     }
...
545     tptr = update_traceback (current, inet);
546     free_heap_data (current);
547   }
...
552 }
```

Figure 4-20: A partial snippet of the `route.c` file in the VPR Benchmark, showing the main steps of the `route_net` function.

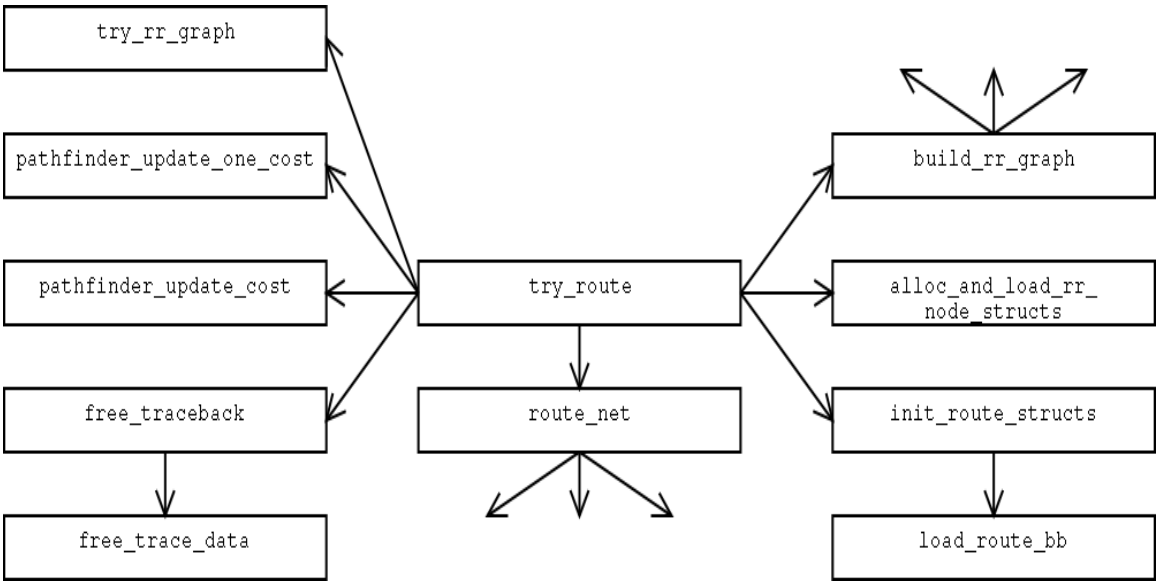


Figure 4-21: The subtree dominated by `try_route` in the VPR benchmark, omitting the subtree dominated by `build_rr_graph` and `route_net`.

snippet from the source code. This function resembles the Dijkstra's algorithm, and thus corresponds to lines 5-12 in Figure 4-15, although the order of the statements are slightly different because the pseudocode uses a repeat-until loop and the implementation uses a while loop. Lines 508-509 in the source code initializes the heap and reads its minimum for the first time, corresponding roughly to lines 6 and 8 in the pseudocode. Line 530 find the neighbors of the nodes with known distance and adds the neighbors to the heap. This line corresponds to lines 9-11 in the pseudocode. Lines 532-533 removes the head (minimum) of the heap and reads the next one, similar to line 8 in the pseudocode.

Figure 4-22 shows the snippet from the source code. Figure 4-21 shows the subtree dominated by `try_route`. We omit the subtree dominated by `net_route` because this subtree is shown in Figure 4-19. We also omit the subtree dominated by `build_rr_graph`. This function is primarily concerned with building the graph from the input description (line 272 in the pseudocode), and thus is not interesting for our discussion.

Line 300 in the code corresponds to line 1 in the pseudocode, and line 304 in the code corresponds to line 2 in the pseudocode. Line 307 in the code updates the

```
239 int try_route (...) {
...
272   build_rr_graph(...);
...
300   for (itry=1;itry<=router_opts.max_router_iterations;itry++) {
301
302     pathfinder_update_cost (pres_fac, router_opts.acc_fac);
303
304     for (inet=0;inet<num_nets;inet++) {
305       if (is_global[inet] == 0) {       /* Skip global nets. */
306
307         pathfinder_update_one_cost (...);
308         is_routable = route_net (inet, router_opts.bend_cost,
309                                  router_opts.route_type);
...
321       }
322     }
...
341 }
```

Figure 4-22: A partial snippet of the `route.c` file in the VPR Benchmark, showing the main steps of the `try_route` function.

cost a specific net, and corresponds to line 4 in the pseudocode. As we see in the discussion above, lines 5-12 in the pseudocode corresponds to the call to `route_net`, as shown in lines 308-309 in the pseudocode. Lines 3,13-16 do not correspond to any lines in this function because those steps are mostly rearranging the pointers from the routing tree and updating the distance. From the internal representation of the VPR benchmark, these steps are not urgent and could be postponed until the actual "distance" is needed.

The LogiView methodology produces a dominator tree and a list of phase-markers. The dominator tree provides an organizational view of the functions in the program, and the phase-markers identify which functions in the dominator tree mark the high-level structures. By examining the phase-markers and the dominator tree, one can infer various informations about the program structure and logically organize the functions according to their task.

# Chapter 5

# Function Names Analysis

The LogiView methodology does not depend on function names in order to generate the dominator tree and identify the phase-markers. However, in our discussion in the previous chapter, we rely on the semantic meaning behind these names to speculate and infer the task of each subtree. Thus, there is an implicit correlation between the names of the functions, their position in the tree, and the behavior of the subtree. This chapter provides a methodology to formulate that correlation and automatically detect such correlation when it exists.

The rest of this chapter is laid out as follows. First we will give an example that illustrates how function names provide semantic information about their functionality. Then for each program, we build a *profile* of the program, that is a list of names that are most relevant to that program. We also introduce a formula to quantify the relevance of these names to the program. Then, given several programs that have the same functionality, we introduce a formula to compare their profiles and extract the names that are most relevant to the functionality. These names form a *dictionary* for the given class of programs. Finally, we provide another formula to measure the likeliness that a program belongs to a certain class by comparing the program profile and the class dictionary.

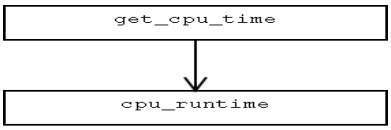


Figure 5-1: A section of the dominator tree from the zChaff benchmark.

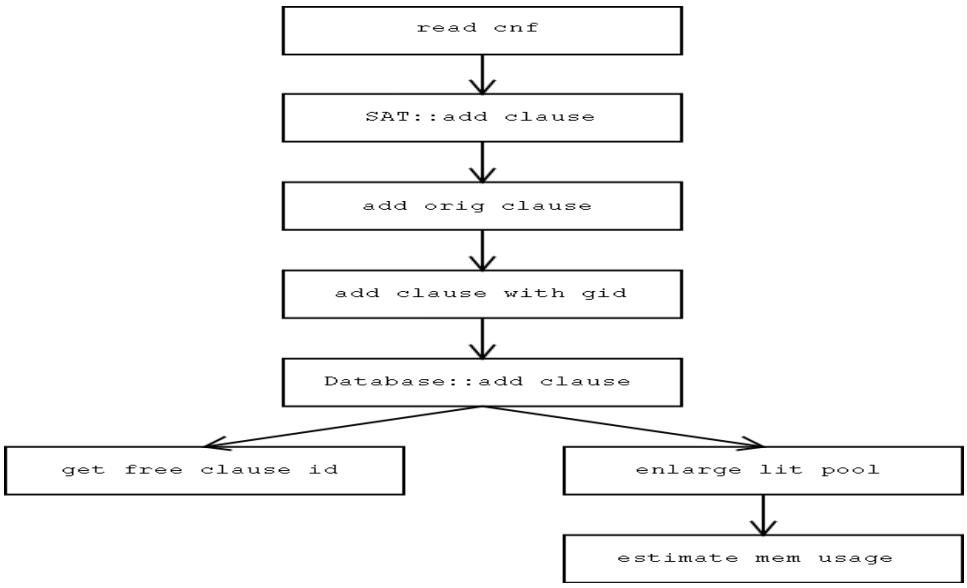


Figure 5-2: A section of the dominator tree from the zChaff benchmark.

## 5.1 Overview

Typically programmers do not name their functions randomly, but instead they chose names that have semantic meaning that suggest the functionality of the program

Figure 5-1 shows a section of the dominator tree fom the zChaff benchmark [9]. It is a benchmark that solves the satisfiability problem by trying to find a satisfying assignment to a boolean formula in Conjunctive Normal Form. Without examining the source code, one can guess that this particular subtree is responsible in retrieving the CPU runtime in order to determine how much time the program spends in a loop or routine. This guess is only based on the function names and the fact that `ge_cpu_time` dominates `cpu_runtime`.

A more complex example is shown in Figure 5-2. The function `add_clause` is responsible for adding a clause to the boolean formula, while making sure the addition

of the clause is consistent with the current representation in the database. In order to add a clause, first the database has to find a "free clause id", that is a clause ID that is not used by other clauses. Then, the database may expand the required memory if the current memory usage is insufficient. Again, these guesses can be made without looking at the source code, and only relying on the names of the functions as well as the (partial) structure of the tree.

We know from the frequencies in the dominator tree that the function `read_cnf` is called only once during the execution of the program, and that the other nodes are called roughly $N$ times each. Then we can make further guesses that the function `read_cnf` opens the input file that contains the CNF (Conjunctive Normal Form) formula, that there are roughly $N$ clauses in that formula, and that each clause invokes a call to `add_clause`. Thus, a knowledge of the frequencies of the nodes yields additional information about the functionalities of each subtree in the dominator tree.

For our analysis, we leverage the LogiView methodology to find the phase-markers. LogiView relies on the structure of the dominator tree and the number of calls to each node (the frequency of the nodes). We shall then combine the result from the LogiView algorithm with names analysis.

## 5.2 Name Splitting

First, we split each function name into a collection of tokens. Each token is a word that the programmer possibly had in mind when he chose to name the function a certain way. For example, we want to split the name `This_is_MyFunction` into (`this`, `is`, `my`, `function`). Tokens are always represented in lower-case letters, so tokens `my` and `My` are considered the same, and they both are stored as `my`.

Names that are separated by lower-case and upper-case letters are separated according to the case change, but a token may consist entirely of upper-case letters. For example, `DatabaseSolver` becomes (`database`, `solver`), and `CSolver` becomes (`c`, `solver`), but `BCP` becomes (`bcp`).

Names that are separated by delimiter characters are separated as the delimiter

separates them. For the purpose of this thesis, the only delimiter character is the underscore ("_") For example, `optimize_svm` becomes (`optimize`, `svm`).

Characters that are not alphabets are discarded. For example, the name `Solver::add_clause3` becomes (`solver`, `add`, `clause`).

If a name is separated by both delimiter characters and letter case difference, then the name is tokenized according to delimiter characters first, followed by case differences. In general, we don't expect programmers to use a mixed method of naming.

## 5.3 Application Profile

The LogiView algorithm shown in Figure 3-10 separates the functions into disjoint groups. For each token, we assign a *frequency*, that is the number of groups the name occurs in. A *profile* of an application is a function that maps each token to its frequency. Furthermore, each token frequency is normalized by the maximum token frequency in the application. Formally, the profile of application $A$ is $P_A$ : String$\rightarrow R$, and $P_A(s) = \frac{n}{M}$ if $s$ occurs in exactly $n$ groups and $M$ is chosen such that $\max_s P_A(s) = 1$

The profile identifies the most relevant token to an application. Tokens that are most related to the application will appear in many functions and groups, and thus have greater frequency. For example, the token "clause" is ubiquitous in the zChaff benchmark. The CNF formula is composed of clauses and most approaches to solving the satisfiability try to resolve one clause at a time. Thus many function names contain the word "clause", and we expect that the token clause has frequency 1 or almost 1.

The frequency of each token is proportional to the number of groups it occurs in, not the number of functions it occurs in, so we don't over-reward chain patterns and leafy patterns. When a task is delegated from one function to another, these functions form a chain pattern and they often have similar names. As in Figure 5-2, the task of adding a clause is delegated from `Solver::add_clause` to `add_orig_clause` to

`add_clause_with_gid` to `Database::add_clause`. While all the nodes in this chain perform only one task of adding the clause, there are at least four functions with the word "add" and "clause" in their names. In our current scheme, the word "add" and "clause" are only counted once in this group instead of four times.

The profile is normalized so that we don't punish nor reward applications with many or few groups. An application $A$ may have a few groups but with names that really suggest its functionalities, Thus the tokens that are most meaningful in this application occur only a few times at most, because they can occur at most once in each group. On the other hand, another application $B$ may have many groups with names that are just as rich semantically. The most meaningful token in $A$ may occur much less times as the most meaningful token in $B$ simply because $B$ has more groups. Thus without normalizations, $B$ would seemingly have more correlation with its function names because the frequencies are high, while $A$ suffers from the opposite effect. But because we normalize so that both of these tokens have frequency 1, We neither punish $A$ or reward $B$ for having too few or too many groups.

We also avoided normalizations with regard to the sum. When we normalized each profile so that the sum of the frequency is 1, we over-punish the applications which have many distinct names. Consider an application $A$ whose profile prior to normalization is $(100, 50, 10, 2, 1, 1, 1, \ldots, 1)$ where there are 38 ones. The sum of the frequency is 200. If we normalized such that the sum of the frequency is 1, the resulting profile would be $(0.5, 0.25, 0.05, 0.01, 0.005, \ldots, 0.005)$. Suppose another application $B$ has profile $(100, 1)$ before normalization. The sum of the frequency is 101, so if we normalized according to the sum, the profile of $B$ would be $(0.99, 0.01)$. Seemingly, $A$ has less correlation with its names than $B$ because the highest frequency in $B$ is higher than all frequencies of $A$. But this disrepancy is not because $B$ has more meaningful names. Both applications have a token that occurs 100 times. The tokens that occur once in $A$ could helper functions or generic names (`main`, `malloc`, etc) that are not essential to $A$'s functionality. Having a few or many tokens with small frequencies should not affect the highest token's frequency in the application. Therefore, we chose our normalization scheme such that the highest frequency is

always 1.

## 5.4 Class Dictionary

A *class of applications* are a collection of applications that are written for the same purpose. Often but not always, they have identical input and output format. They may be different implementations of the same algorithm, or they may use different algorithms entirely. An example of a class of application is the Satisfiability Solvers Class, which embodies applications that solve the boolean assignment satisfiability problem, also known as SAT. zChaff is an example of an application in this class.

Given a class of applications, we wish to build a dictionary that lists the names that are most related to the task of the applications in the class. For example, we expect that SAT solver implementations will include tokens such as "solver", "clause", and "literal" with high ranks in their profiles. We want our dictionary to recognize that these words are indeed related to the task of solving the satisfiability problem.

## 5.5 Dictionary Construction

Given a class of $k$ programs $A_1, A_2, \ldots, A_k$ , we build a *dictionary* for this class. A dictionary is a function that maps each token to a numerical value, such that the names that are more meaningful and more associated with the class have higher values. I define the dictionary of the class of application $\{A_i\}$ is the function $\mathcal{D}$ : String$\rightarrow R$, such that

$$\mathcal{D}(s) = \frac{\sum_{i \neq j} \sqrt{P_{A_i}(s) P_{A_j}(s)}}{\frac{1}{2}k(k-1)}$$

For example, if the class has three applications in it, and the word "solver" has frequency $0.9, 0.8, 0.75$ in their profiles respectively, then the value of "solver" in the dictionary is:

$$\frac{\sqrt{0.9 \times 0.8} + \sqrt{0.9 \times 0.75} + \sqrt{0.8 \times 0.75}}{\frac{1}{2} \times 3 \times 2} = 0.8149$$

The value of a name is non-zero only if the name occurs in at least two profiles, because each term $\sqrt{P_{A_i}(s)P_{A_j}(s)}$ requires contribution from two profiles. Thus, the names that only occur in one application do not contribute to the dictionary. Usually these are names that are related to or stand for the person or place where the application is written, such as "haifa" in "HaifaSat"[41] or "jeru" in "JeruSat" [28] (both are city names where the applications were developed).

Conversely, if a name occurs in at least two applications, the value is immediately positive. Each term in the summation is non-negative, and when two profiles have positive value for a name, the product from those two profiles will be positive. Moreover, a name that occurs in more profiles is rewarded quadratically due to the nature of pairwise summation. If a name occurs in $m$ profiles out of $k$ profiles, then exactly $\frac{1}{2}m(m-1) = O(m^2)$ terms in the sum will be positive.

The use of square roots ensures that the contribution from a pair always lies between the name's frequency in each profile. If $x \geq y \geq 0$, then $x \geq \sqrt{xy} \geq y$. Thus, the contribution from two profiles to the dictionary is the geometric average of the values of the names in each profile.

The $\frac{1}{2}k(k-1)$ normalization factor ensures that classes with many applications are not rewarded more than those with few applications. There are exactly $\frac{1}{2}k(k-1)$ summands in the summation, thus we are in essence taking the arithmetic average of pairwise geometric average of each of the profiles.

The value of each name will be between 0 and 1 inclusive. In order for a name to have a value 1, it must have the highest frequency in all the profiles. In other words, it has to be the most occuring name for all the applications in the class. Based on my empirical analysis, it is very rare for a name to have a value 1, but it is not impossible.

## 5.6 Punishing Generic Names

The dictionary often contains names with high values that are not necessarily related to the functionality of the applications. These are names that are commonly used by programmers such as "get", "set", "create", etc. The presence of these names

overshadows actual important names in the program. One way to recognize these generic names is by comparing accross multiple dictionaries. Because these names are commonly used in any programs regardless of their functionalities, we expect generic names to have high values in multiple dictionaries. We introduce a punishment function that quantifies how common a name is from comparison accross multiple dictionaries.

A *punishment function* is a function that maps each name to a numerical value. We wish the value to be high when a name is common and it occurs in many dictionaries with high values. If $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_k$ is a collection of $k$ dictionaries, the punishment function $\mathcal{T}$ : String $\rightarrow R$ is defined as

$$\mathcal{T}(s) = \sum_{i \neq j} \sqrt{\mathcal{D}_i(s)\mathcal{D}_j(s)}$$

The punishment function formula resembles the dictionary computation formula. Thus we infer the following properties of the punishment function.

1. The value of a name is positive if and only if the name is shared by two or more dictionaries. Names that are specific to one dictionary will have a punishment value of zero. Thus, tokens that are suggestive of the functionality of a class (e.g. "sat") will have punishment value close to of zero because it is unlikely for that token to occur in other classes.

2. Due to the nature of pairwise summation, the punishment of tokens that are shared by multiple dictionaries are increased quadratically.

If a name has high value in one dictionary but small values in others, then its punishment value will be small too. For example, suppose a token has value 0.9 in one dictionary, and it has values 0.01 and 0.005 in other dictionaries. This scenario could happen if this token is highly correlated to a particular class, but also weakly correlated (but not zero) to some other classes. Then its punishment value would be:

$$\sqrt{0.9 \times 0.01} + \sqrt{0.9 \times 0.005} + \sqrt{0.01 \times 0.005} = 0.3741$$

A special case of this property is when the value is positive in one dictionary and zero in other dictionaries. Then the punishment value would be zero, and it reflects the fact that this token is not "shared" at all.

The punishment value of each token is between 0 and $\frac{1}{2}k(k-1)$ inclusive. We don't normalize by requiring the value to be between 0 and 1 for the reasons that will become clear in the next section. In the previous sections, we normalize the profiles and dictionaries because we eventually compare values accross multiple profiles and dictionaries. But since there is only one punishment function, the presence or lack of normalization does not matter in our methodology.

## 5.7 Application Score

An application's profile indicates which tokens are the most related to the application's functionalities, while a class' dictionary describes which tokens are the most related to the task of the applications in that class. Thus, given a candidate application $A$ and a class of applications $\mathcal{C}$, we want to know if $A$ belongs to $\mathcal{C}$ or not by comparing $A$'s profile and $\mathcal{C}$'s dictionary.

One sign that $A$ belongs to $\mathcal{C}$ is if $P_A$ is similar to $\mathcal{C}$'s dictionary. The tokens that have high values in one should also have high values in another and vice versa. But we also want to minimize the effect of the generic names, thus we have to make use of the punishment function that we developed earlier. We introduce a formula to compute the similarity between $A$ and the class $\mathcal{C}$.

Given a candidate program $A$ and a class of applications $\mathcal{C}$, the *Similarity Score* $f(A, \mathcal{D}_\mathcal{C})$ measures the closeness of $A$ to $\mathcal{C}$. The higher the score is, the more likely $A$ belongs to $\mathcal{C}$. If $\mathcal{D}_\mathcal{C}$ is the dictionary of class $\mathcal{C}$, the score is defined as

$$f(A, \mathcal{D}_\mathcal{C}) = \sum_s \frac{P_A(s)\mathcal{D}_\mathcal{C}(s)}{e^{T(s)}}$$

where the summation is run over all possible tokens. In practice, we only need to consider the union of the tokens in $A$ and $\mathcal{D}_\mathcal{C}$, because other terms will be zero. Each

name that is shared between $A$'s profile and $\mathcal{D}_\mathcal{C}$ contributes one positive term to the sum. This contribution is high if both $P_A(s)$ and $\mathcal{D}_\mathcal{C}(s)$ is high. It means that if a token occurs frequently in application $A$, and that token is strongly associated to the class $\mathcal{C}$, it is a supporting evidence that $A$ belongs to $C$.

The names that are shared by multiple dictionaries are punished exponentially. When a token has a punishment value of zero, $T(s) = 0$ and thus the term doesn't suffer punishment at all. But when $T(s)$ is high, the token's contribution to the sum decreases exponentially.

When we have a candidate application $A$, and several classes $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$, we compute the similarity score of the application against each class. We choose the class that has the highest similarity score, and we say that $A$ most likely belongs to that class.

One way to use this formula is when we have a collection of application classes, and we have a candidate application. If we know that the candidate must belong to one of the classes in our collection, we can calculate the similarity score between the candidate application and each class. The class that has the highest similarity score is the most likely class where the candidate application belongs, and thus we *classify* the candidate application into that class.

There are a number of possible applications of this methodology. For example, a system can invoke an end-to-end optimization if it knows the application is in a certain class. Different classes may have different optimization techniques, and by identifying the class correctly, the system can choose the most appropriate optimization. Another example is when the system detects that one of its modules belongs to a certain class. If the system has alternative implementations for the same class, it can suggest substituting the module with another module with the same functionality. Also, if the system detects that two of its modules have the same functionality, it can suggest a refactorization of these two modules.

## 5.8 Application Class Analysis

We evaluate five classes of applications. First, we describe 5 application classes and their dictionaries. Then we show the generic tokens that occur in multiple classes with their punishment function values. Finally, we performed a leave-one-out test. In this test, we remove one application, recompute the dictionaries and the punishment function, and classify that application into the class that has the highest similarity score. We do this for each application and show that most applications are classified into the actual class where they belong.

The application classes that we examine are:

1. Satisfiability Solver (SAT)
2. Support Vector Machine Learning (SVM)
3. Deterministic Finite Automata Learner(DFA)
4. Linear Programming Solver (LP)
5. Fast Fourier Transform Calculator (FFT)

These classes are chosen due to the growing importance in the field of machine learning, optimization problems and scientific computing.

### 5.8.1 Satisfiability Solver

A SAT Solver takes as input a boolean formula in Conjunction Normal Form, and outputs whether there is an assigment of variables that makes the formula true. This problem is known to be in NP-complete. The applications in this class are mostly written to compete for the best implementation and performance for the SAT Competition [31]. We have six applications in this class: zChaff [9], Jerusat v1.2 [28], Jerusat v1.3 [28], Minisat [7], HaifaSat [41], and MarchDL [12].

The SAT problem itself is NP-complete and several heuristics have been found to speed up the search. All the benchmarks above use different algorithms and heuristics, except Jerusat v1.2 and Jerusat v1.3 which are different versions by the same author.

| Token $s$ | Dictionary Value $\mathcal{D}(s)$ |
|---|---|
| get | 0.435677 |
| clause | 0.40127 |
| lit | 0.252596 |
| add | 0.231888 |
| solver | 0.22264 |
| var | 0.209158 |
| io | 0.199065 |
| new | 0.194051 |
| free | 0.158784 |
| to | 0.146992 |
| in | 0.14346 |
| libc | 0.141088 |
| default | 0.120695 |
| file | 0.119236 |
| int | 0.10032 |
| internal | 0.0970132 |
| init | 0.0961564 |
| sat | 0.0936139 |
| read | 0.0692947 |
| cl | 0.0587285 |

Table 5.1: The dictionary for the SAT application class.

They use the same algorithm with different parameters. All benchmarks are run with the same input, the Airport Scheduling Problem ([9] `p05.cnf`).

Table 5.1 is the resulting dictionary for the SAT class, sorted according to their values $\mathcal{D}(s)$. We only display the top twenty tokens. The topmost token is a generic token ("get"), but the next five tokens have semantic correlations with SAT ("clause", "lit", "add" "solver", "var"). "Clause", "lit", and "var" are ranked up high because the CNF form is a conjunction of clauses, each of which is a disjunction of literals. These literals are variables that must be assigned a true or false value.

"Add" is a mix between generic token and meaningful token. There are function names such as `add_binary_equivalence` and `root_add_constraint_resolvent` in MarchDL and `add_clause` in HaifaSat and zChaff that really associate "add" with SAT. But there are also names such as `exchange_and_add` and `atomic_add` in HaifaSat that are quite general and do not strongly suggest the SAT class.

| Token $s$ | Dictionary Value $\mathcal{D}(s)$ |
|---|---|
| svm | 0.564235 |
| kernel | 0.287457 |
| cache | 0.275783 |
| compute | 0.136083 |
| example | 0.117851 |
| get | 0.096225 |
| solve | 0.096225 |

Table 5.2: The dictionary for the SVM application class.

"Sat" is surprisingly ranked quite low in Table 5.1. The token stands for "satisfiability" and thus we expected that many function names would contain this token. But in reality, only zChaff, MarchDL, and Minisat have this token. Even then, MarchDL only has one function with that token (`transformTo3SAT`). zChaff has "sat" in multiple functions (`SATSolver`, `SolveSAT`, etc), but they all fall into one group, thus the frequency in the zChaff profile was also low.

### 5.8.2 Support Vector Machines Learning

The Support Vector Machines (SVM) trainers take as inputs two sets of points in a multidimensional space and try to find maximum separation of those points. A separation is a plane such that all points in one set are in one side of the plane and all points in the other set are in the other side of the plane. The separation is represented as a plane in a higher dimension. The trainer's objective is to find the plane such that the distance from the points to the plane is maximum. If perfect separation is not possible (if there is no plane such that all points are in their respective side), then the trainer maps these points into a higher dimension where the separation becomes possible. The coordinates of the points in the new dimensions are determined by a *kernel* function [4], which varies for each algorithm.

Our benchmarks are downloaded from [35]. For our experiments, we used three applications: SVM Light [15], GiniTrain [3], and SVM Torch [4].

Table 5.2 shows the dictionary for this class. There is a generic token "get" but it ranks quite low in the dictionary. The top two tokens ("svm" and "ker-

nel") are very specific to SVM. The token "cache" ranks third. Although this token is not very specific to SVM, it occurs in many functions within the SVM context. In Gini Train it occurs in `kernel_insert_cache` and in SVM Light it occurs in `touch_kernel_cache` and these functions suggest the SVM class. However, "kernel" also occurs in `block_evaluate_cache` which is quite generic and does not suggest the SVM class.

### 5.8.3 Deterministic Finite Automata Learner

The applications in the DFA class learns the strings in the language of a Deterministic Finite Automata, given the known accepted and rejected strings. They are all written to compete in the Abbadingo Competition [18]. The task of the competition was to predict whether an unknown DFA would accept or reject the test strings. The learners are provided with a set of training strings that had been labeled as accepted or rejected.

In our experiments, we used three applications which are variations of the same algorithms [16] and all written by the same person: Red Blue Simulator [17], Traxbar Simulator [17], and RLB Simulator [17].

The algorithm is called Evidence Driven State Merging (EDSM) [16]. It attempts to "guess" the general structure of the known accepted strings by merging similar strings together. In order to achieve the maximum probability of correct guesses, this algorithm uses Augmented Prefix Tree Acceptor (APTA), a tree where each node corresponds to one symbol in the string. The paths from the root to the nodes represent the accepted strings, so the number of nodes is equal to the number of the accepted strings.

The algorithm merges two nodes if they are "similar" enough (if the subtrees of these two nodes share many labels). The method to calculate the similarity score between two subtrees differs from one algorithm to another. Although the merged nodes will have similar subtrees, these subtrees are not exactly the same. Thus after the nodes are merged, these two subtrees need to be "resolved". The resolution method of the subtrees under the merged nodes is also different for each algorithm.

| Token $s$ | Dictionary Value $\mathcal{D}(s)$ |
|---|---|
| itable | 0.929618 |
| pqueue | 0.822109 |
| make | 0.560456 |
| set | 0.409254 |
| test | 0.409254 |
| my | 0.28635 |
| push | 0.28635 |
| plist | 0.277778 |
| pstack | 0.272166 |
| empty | 0.227688 |
| pop | 0.227688 |
| and | 0.225492 |
| augment | 0.177272 |
| lookup | 0.177272 |
| malloc | 0.177272 |
| mbtowc | 0.177272 |
| memset | 0.177272 |
| tree | 0.177272 |
| deallocate | 0.175682 |

Table 5.3: The dictionary for DFA application class.

Once all the similar nodes are merged, the resulting tree represents the most likely structure that generates the given accepted strings.

Table 5.3 shows the top twenty tokens for the DFA class. We see that two of the top tokens are very generic ("make" and "set"). However, there are many tokens that occupy the table suggest stack and queue operations ("itable", "pqueue", "push", "plist", "pstack", "empty", and "pop"). We also see "augment" and "tree" in the list although they do not rank very high. This table suggests either that the augmented prefix tree is implemented with stacks and queues, or the stacks and queues are heavily used in the tree-related algorithms.

Function `make_breadth_first_node_list` in RLB traverses the tree in a breadth-first traversal. It uses a queue to store the nodes as the tree is being traversed ([17] `rlb.c` line 336-367). Function `accept_the_node` in Traxbar detects when a node cannot be merged anymore and it "accepts" the node to be a state in the string generator. Because these nodes must be accepted one at a time, the function uses a

queue to store the nodes. ([17] `traxbar.c` line 546-564). In Red-Blue, the function `walk_graph_fixing_depths_and_backpointers` corrects the fields of the nodes that may be rendered incorrect during the merges. This function also visits the the tree using a breadth-first traversal and store the nodes in a queue. ([17] `traxbar.c` line 386-422).

The dictionary for the DFA class is a desirable dictionary. Two highest ranked tokens have very high $\mathcal{D}(s)$ value (0.93 and 0.82 respectively). These tokens are not generic tokens. Although they do not semantically suggest Deterministic Finite Automaton or inference learning, they are associated with queues and trees, and consequently with breadth-first traversal of a tree. This traversal mechanism is used many times in the EDSM algorithms.

### 5.8.4 Linear Programming Solver

The applications in the Linear Programming Solver (LP) class perform simplex method on multidimensional spaces. The input is a set of linear constraints (hyperplanes) and a linear objective function. The solver outputs whether the problem is *feasible*, whether there is a point that satisfies all the constraints. If the problem is feasible, the solver also outputs the point that maximizes the objective function.

The simplex method chooses any point on the intersection of two constraint hyperplanes. At each step, the method chooses the direction that would increase the objective function the most, and translates the chosen point in that direction. The distance is determined such that all constraints are still satisfied after the translation. This step is repeated until the objective function does not increase anymore. When the constraint coefficients are represented in a matrix, the simplex method is equivalent to a series of matrix transformations that can be performed very efficiently.

Although many algorithms have been developed to identify stopping condition as early as possible, this problem is still an open research field. For our experiments, we used six benchmarks that are downloaded from [25]. They are Soplex [25], QSolver [25], GLPK [25], CLP [25], LPabo [25], and LPako [25]. We run them with the same input data consisting of 21 constraints in 53 dimensions ([25] `hello.mps`).

| Token $s$ | Dictionary Value $\mathcal{D}(s)$ |
|---|---|
| add | 0.125966 |
| create | 0.0993527 |
| mat | 0.0938832 |
| to | 0.0890972 |
| set | 0.0760733 |
| elem | 0.0666667 |
| hash | 0.0666667 |
| insert | 0.0525783 |
| free | 0.051957 |
| calc | 0.0333333 |
| hashkey | 0.0333333 |
| search | 0.0333333 |
| get | 0.0267602 |
| init | 0.0234215 |
| name | 0.0221487 |
| col | 0.021171 |
| key | 0.019983 |
| int | 0.0190665 |
| row | 0.0172189 |
| io | 0.0168207 |

Table 5.4: The dictionary for the LP application class.

| Token $s$ | Dictionary Value $\mathcal{D}(s)$ |
|---|---|
| timer | 1 |
| doit | 0.5 |
| start | 0.5 |
| stop | 0.5 |
| bench | 0.333333 |
| malloc | 0.333333 |
| fft | 0.1 |

Table 5.5: The dictionary for FFT application class.

Table 5.4 shows the dictionary for the LP class. The top tokens are very generic names ("add", "create", "to", and "set"). Although the third-ranked token "mat" stands for "matrix", it does not specifically suggest the simplex method either. There are many matrix-related algorithms that are different from the simplex method. The next non-generic token is "row" that is ranked quite low in the table. This token not only stands for matrix rows, but also for the technical term "row" in linear programming that means an inequality constraint. Each inequality constraint is worth one row and each equality constraint is worth two rows.

This dictionary is rather undesirable because the tokens that semantically suggest the simplex method are ranked low ("mat" and "row"). Moreover, the $\mathcal{D}(s)$ values for all tokens in this dictionary are generally low. The highest value is only 0.125 compared to 0.4 in SAT (Table 5.1) and 0.5 in SVM (Table 5.2), and this highest value is attained by a generic token ("add"). The scarcity of the class-specific tokens and the low values for all token show that this application class does not have a very cohesive function names within class. There is no strong naming convention among the simplex method programmers.

### 5.8.5 Fast Fourier Transform

The applications in the FFT class performs Fast Fourier Transform to the given inputs. There are six applications in this class: Arprec [8], Bloodworth [8], FFTE [8], Cross [8], DFFT [8], and Harm [8].

Table 5.5 shows the dictionary for the FFT class. This dictionary is also undesir-

able for different reasons than that of the LP class. The tokens in the dictionary don't strongly suggest the task of performing s Fast Fourier Transform, except "fft" which ranked the lowest in the dictionary. This phenomenon is because each benchmark implemented the transformation in only one or two functions, and those functions are named differently for each benchmark.

For example, in Arprec there are several functions that perform the Fourier transformation. But these functions are responsible for different types of inputs (complex or real, integer or float, power of two or not, etc) and are optimized differently. Thus, for each execution of the program, only one of these functions appears in the call graph and in the dominator tree. These functions are named such as `mp_real::mp_init` and `mp_complex::mp_init`. In the Bloodworth benchmark, the substantial part of the transformation is performed in only one function. Like Arprec, although there are several functions, but these functions are responsible for different types of inputs. Moreover, these functions are named very specific to Bloodworth, such as `Bloodworth_Q2_FwdFFT` and `Bloowdwoth_Q2_RevFFT`.

As a result, the dictionary in Table 5.5 contains only six tokens because it only shows the names that are not benchmark-specific. The token "fft" surprisingly ranks the lowest in that table. The highest ranking token is "timer" which has a $\mathcal{D}(s)$ value of 1, which means that this token is the most occuring token in all six benchmarks. Because these benchmarks are all written for benchmarking purposes (to compare the performance of different implementations of FFT for different types of inputs) [8], each benchmark has a timer interface that measures the execution time for each input.

The undesirable dictionaries for both the LP and FFT class show that not all application classes have a strong naming convention. Even for a very well-known algorithms such as the simplex method and the fast fourier transform, programmers choose to name their functions differently. Although these names may be semantically rich by themselves, the names vary from one program to another. Because our methodology that relies on the names uniformity within each class, the resulting dictionaries suffer from the lack of strong naming convention.

### 5.8.6 Punishment Function

From the dictionaries of these 5 classes, we compile the list of shared tokens and compute the punishment function for each token. Table 5.6 lists the top 20 tokens in the punishment function. The values of a token's punishment ranges from 0 to $\frac{1}{2}k(k-1)$. Because we use $k = 5$, then the greatest possible value for a punishment function is 10. We see that the top tokens in this list are indeed common words that programmers often use to name their functions.

The token "solve" is among the top twenty tokens in Table 5.6 because this token appears in the dictionary for SVM and LP ($\mathcal{D}_{SVM}(s) = 0.096$ and $\mathcal{D}_{LP}(s) = 0.004$).

The token "push" has a punishment value 0.07. This token appears in the dictionary for SAT and DFA ($\mathcal{D}_{SAT}(s) = 0.021$ and $\mathcal{D}_{DFA}(s) = 0.286$). Although the dictionary value for DFA is quite high, because the dictionary value for SAT is small then the punishment value is also small as expected. Thus, despite the fact that this token is shared between SAT and DFA, SAT only uses is lightly while DFA uses it heavily. As a result, the punishment value is small because this token still suggests DFA more than it suggests SAT.

### 5.8.7 Leave-One-Out Test

In order to test that our methodology successfully builds the dictionaries that are most relevant to each class, we run a test procedure for all the benchmarks, called the leave-one-out test. For each application $A$, we delete $A$ from its application class and rebuild the dictionary for $A$'s class without using $A$'s profile. Then we recalculate the punishment functions with this new dictionary. We also compute the similarity score of $A$ against all the dictionaries, and we classify $A$ to the class with highest similarity score. We call the classification of $A$ a "success" if $A$ is classified to the correct class from which it was deleted, and "failure" otherwise. This procedure mimics the classification procedure in the sense that, if $A$ had been a candidate application and we don't know which class it belongs to, then $A$ would be classified in the same manner as our leave-one-out test method.

| Token | Punishment Value $\mathcal{T}(s)$ |
|---|---|
| malloc | 0.411743 |
| get | 0.363472 |
| set | 0.316481 |
| free | 0.245363 |
| add | 0.170909 |
| to | 0.11444 |
| and | 0.113238 |
| empty | 0.104808 |
| push | 0.0767548 |
| io | 0.0578654 |
| create | 0.0530481 |
| init | 0.0474565 |
| int | 0.0437349 |
| dispose | 0.030924 |
| default | 0.0296494 |
| libc | 0.0289664 |
| file | 0.0266288 |
| elem | 0.0261876 |
| mpn | 0.0218693 |
| solve | 0.0210903 |

Table 5.6: The punishment function for the shared tokens.

| Benchmark | $f(A, \mathcal{D})$ with SAT | $f(A, \mathcal{D})$ with SVM | $f(A, \mathcal{D})$ with DFA | $f(A, \mathcal{D})$ with LP | $f(A, \mathcal{D})$ with FFT | Classified as | Success or Failure |
|---|---|---|---|---|---|---|---|
| SAT | | | | | | | |
| Zchaff | 0.7320 | 0.0452 | 0.0039 | 0.0494 | 0 | SAT | Success |
| Jerusat12 | 1.0051 | 0.0486 | 0.0161 | 0.2327 | 0 | SAT | Success |
| Jerusat13 | 1.1981 | 0.0571 | 0.2279 | 0.1831 | 0 | SAT | Success |
| Minisat | 0.5488 | 0.0059 | 0 | 0.0191 | 0 | SAT | Success |
| HaifaSat | 1.0047 | 0.0199 | 0.0978 | 0.0979 | 0.0244 | SAT | Success |
| MarchDL | 1.0047 | 0.0603 | 0.1495 | 0.1790 | 0.0406 | SAT | Success |
| SVM | | | | | | | |
| GiniTrain | 0 | 0.5314 | 0 | 0.0144 | 0 | SVM | Success |
| SVMLearn | 0.1926 | 0.4629 | 0.1128 | 0.0732 | 0.0552 | SVM | Success |
| SVMTorch | 0.2292 | 0.4023 | 0.0994 | 0.0639 | 0 | SVM | Success |
| DFA | | | | | | | |
| RedBlue | 0.0734 | 0.0226 | 2.6786 | 0.0283 | 0.0366 | DFA | Success |
| Traxbar | 0.0721 | 0 | 2.6732 | 0.0477 | 0.0445 | DFA | Success |
| Rlb | 0.0257 | 0 | 2.4754 | 0.0196 | 0.0366 | DFA | Success |
| LP | | | | | | | |
| Soplex | 0.3579 | 0.0277 | 0.1993 | 0.0585 | 0.0952 | SAT | Failure |
| Qs | 0.2927 | 0.0142 | 0.0575 | 0.0364 | 0.0164 | SAT | Failure |
| Glpk | 0.0741 | 0.0237 | 0.0497 | 0.0307 | 0 | SAT | Failure |
| Clp | 0.0124 | 0 | 0.0703 | 0.0102 | 0 | DFA | Failure |
| Lpabo | 0.1985 | 0 | 0.0250 | 0.1192 | 0 | SAT | Failure |
| Lpako | 0.1309 | 0 | 0 | 0.1105 | 0 | SAT | Failure |
| FFT | | | | | | | |
| Arprec | 0.0029 | 0 | 0.0597 | 0.0092 | 2.0011 | FFT | Success |
| Bloodworth | 0 | 0 | 0 | 0 | 1.775 | FFT | Success |
| FFTE | 0.0029 | 0 | 0.0597 | 0.0016 | 2.0261 | FFT | Success |
| Cross | 0.0266 | 0 | 0.0990 | 0.0089 | 2.0261 | FFT | Success |
| Dfft | 0.0029 | 0 | 0.0597 | 0.0016 | 2.0011 | FFT | Success |
| Harm | 0.0029 | 0 | 0.0597 | 0.0016 | 2.0011 | FFT | Success |

Table 5.7: The similarity scores and the classification classes from the leave-one-out test.

Table 5.7 shows the similarity score table for each benchmark against each class. We achieve a 100% success rate in every class except the LP class, where we achieved zero success rate. As we discussed above, the dictionary for the LP class is undesirable due to the lack of cohesive function names among the programs in the class.

However, the programs in the FFT class are all classified successfully despite its undesirable dictionary. Although the dictionary contains tokens that mostly have poor semantic meaning, these tokens are not shared with the other classes. Moreover, the dictionary values for this class are generally high (Table 5.5). In other words, the dictionary for FFT by itself does not intuitively suggest that the programs in the class perform Fast Fourier Transform. But when compared against the other dictionaries, this dictionary distinctively suggests the FFT class compared to the rest. For example, if we look at several dictionaries at once and we are told that one of them is the dictionary for the FFT class, we would be able to identify the correct dictionary for the FFT class. Thus, each program in this class is successfully classified because there is little or no ambiguity between the FFT and the other classes.

The DFA class has the most disrepancy between its own class against other classes. The score of a DFA application to its own class is much higher than those to other classes. This disrepancy may be due to the fact that all three applications are written by the same person, so they are more likely to share the same naming conventions. The disrepancy could also be attributed to the desirable dictionary as we discussed above. The tokens have high dictionary values (Table 5.3), and many of the tokens are meaningful and suggest the breadth-first tree traversal that is used in the EDSM algorithms.

All programs in the LP class are classified unsuccessfully but the programs in the FFT class are classified successfuly, although these two classes both have undesirable dictionaries. We see that the undesirable dictionaries are a result of the lack of strong naming convention among programmers in these classes. Because our methodology relies on the name similarity within each class, the dictionary suffers from such lack of convention. Furthermore, most applications in LP are classified as SAT, which suggests that each of these application is closer to SAT than to one another. This

phenomenon implies that there are a lot of shared tokens between LP and SAT, but the degree of sharing within LP is not as strong as between LP and SAT.

One way to address this problem is by punishing each token differently, according to the desirability of each dictionary. Currently, we have no way to quantify the desirability of a dictionary, but such measure should depend on the composition of the $\mathcal{D}(s)$ values (whether they are high or low values). It should also depend on whether the top tokens are generic or not, by comparing multiple dictionaries. Once we have devised a formula to quantify desirability, we can use it to improve our punishment function so that the tokens from less desirable dictionary are punished less.

# Chapter 6

# Physical Phase Detection

Our phase-markers are intended to encompass high-level structures in the program and mark the boundaries between one logical component to another. These components typically perform very different functionalities, and are likely to have very different physical characteristics throughout their execution. I explore the relationship between the primary data cache miss rates and the phase-markers recognized by LogiView. Specifically, I demonstrate that these phase-markers are useful in the context of not only understanding program structure, but also understanding program performance. Then I also discuss the relationship between LogiView and the traditional phase algorithms that only observe these physical characteristics.

In this chapter, we will examine the behavior of primary data cache miss rates of various benchmarks. We record the miss rates by instrumenting binaries using Program Instrumentation Tool (PIN) [21]. PIN enables us to record every data accesses in the program, and consequently enables us to simulate the cache and keep track the number of cache hits and misses throughout the program execution. After every $k$ accesses, we compute the ratio of the miss to total accesses and plot the ratio. We call $k$ the *window size* of the plot, which can be set arbitrarily. Typically we set the window size so that the entire execution fits within one graph, and individual phase behavior is still apparent and visually recognizable. Later we will see how the choice of the window size affects our results.

Figure 6-1 shows the primary cache data miss rates for an execution of SVM

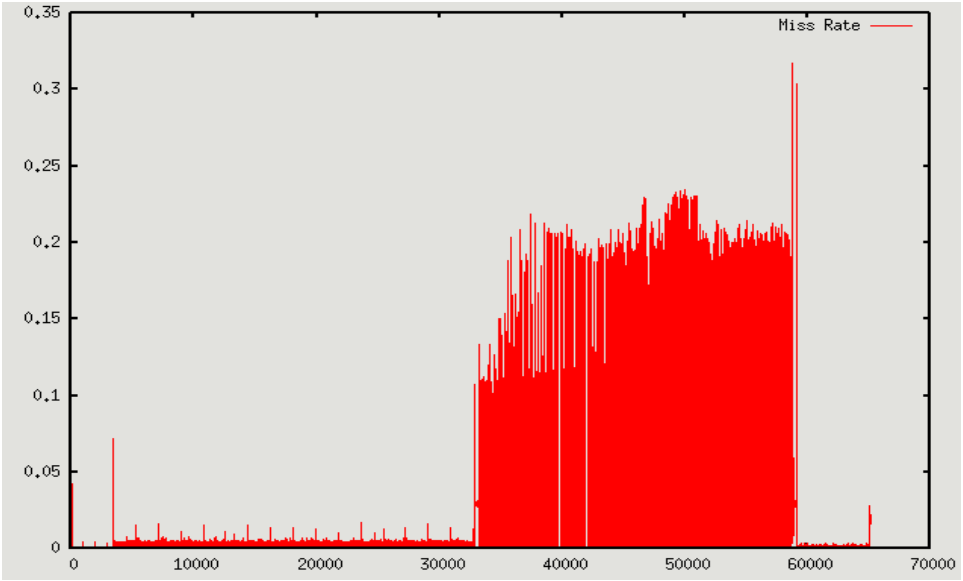


Figure 6-1: Miss rates plot of a benchmark where there are higher miss rates region and lower miss rates region.

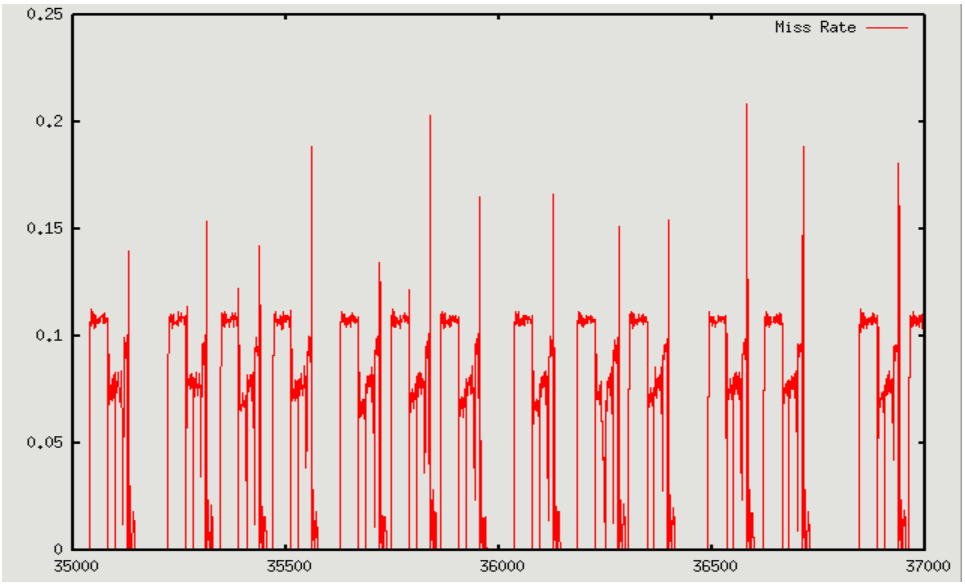


Figure 6-2: A section of miss rates plot where a similar behavior is repeated multiple times.

Light benchmark [15] with window size 2048. The horizontal axis is the "time" of the execution, where each point corresponds to one window. The vertical axis is the miss rate. We can see clearly that there are regions with lower miss rates in the beginning and there are regions with lower miss rates towards the end of the execution. There is an abrupt change of the miss rates between these two regions, which suggests that these regions belong to two different components in the program and they perform different tasks.

Figure 6-2 shows the same plot as in Figure 6-1, but we have zoomed in to one of the higher miss-rates region. We see that there is a pattern that repeats itself multiple times. Most likely, this repetition is caused by a loop whose execution span equals the length of the repeated pattern.

We will show that the phase-markers that we found using LogiView can predict or delineate this change of behavior. Specifically, we observed the following correlation between phase-markers and the change of behavior (miss rates):

1. Each call to a phase-marker marks a region with a repeated pattern of execution. The interval between two consecutive calls to a phase-marker contains a pattern that repeats over every such interval. Figure 6-2 shows an example of a repeated pattern, and each repetition corresponds to one call to a phase-marker.

2. The immediate dominator of the phase-marker (called the phase-parent) encompasses an interval of distinct behavior. The interval between a call to a phase-parent until the exit to the caller function exhibits a behavior that is distinguishable from the interval immediately before and after. Figure 6-1 shows an example where there may be two phase-parents in the program, in which case each of the region corresponds to a call to a phase-parent. There may also be only one phase-parent which corresponds to one of the regions. In this case, we say that the behavior *inside* the call to the phase-parent is distinct from the behavior *outside* the call.

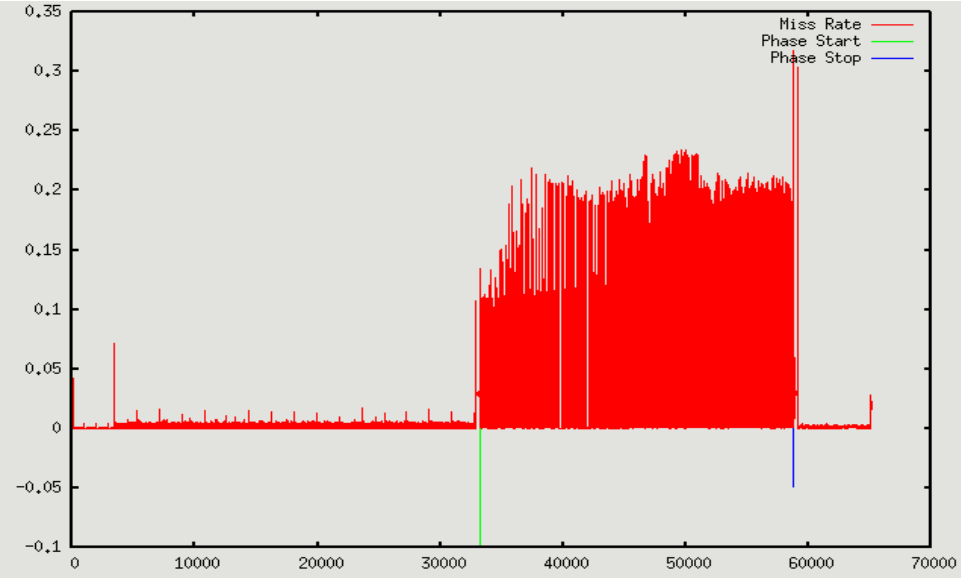


Figure 6-3: The entry and exit to `optimize_to_convergence` in the SVM Light Benchmark.

## 6.1 Results

We run our instrumentation on a number of benchmarks: SVM Light [15], SPEC GNU Zip [36], Timberwolf [38] and Grammar Link Parser [37]. For each of this benchmark, we also use PIN to record every function calls and exits. In each of the plot, the miss rates are indicated by positive values shown in red, and the entry and exit to a function are indicated by negative values of value $-0.1$ and $-0.05$ respectively. If this thesis is available in color, then the entry and exit will also appear in green and blue respectively.

### 6.1.1 SVM Light

We run LogiView on the SVM Light Benchmark [15], with window size 2048. LogiView only presents one phase-marker `optimize_svm` and the corresponding phase-parent `optimize_to_convergence`. These pair are the computational substance of SVM.

Figure 6-3 shows the entry and exit to the phase-parent. There is only one call to this function throughout the entire program execution. The interval between the entry

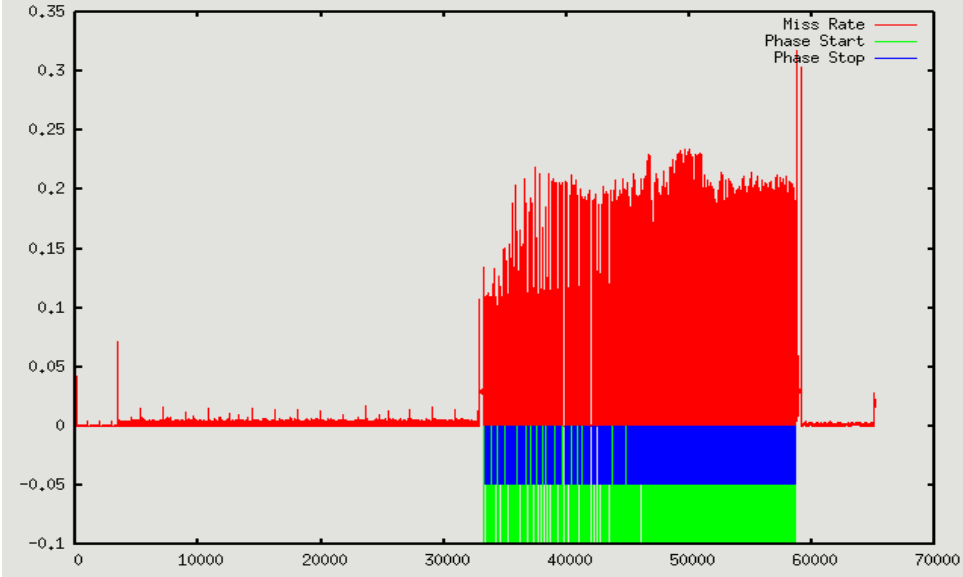


Figure 6-4: Multiple entries and exits to `optimize_svm` in the SVM Light Benchmark.

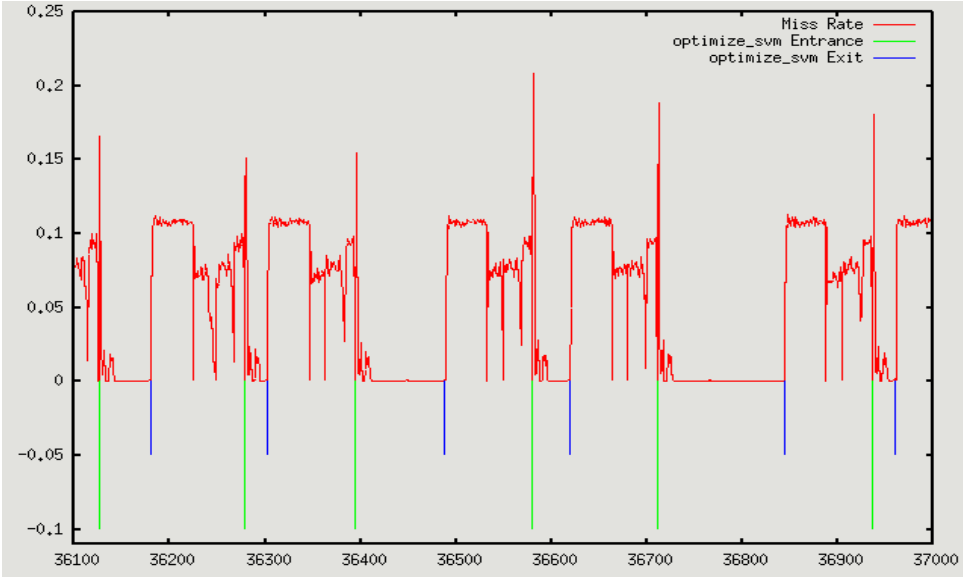


Figure 6-5: A zoomed-in view of Figure 6-4, showing the repetition of the pattern for each call to `optimize_svm`.

```
main {
  // reading and parsing input files
  ...

  svm_learn_classification();

  // writing and printing output
  ...
}

svm_learn_classification {

  // checking data sanity and consistency
  ...

  // initializations
  ...

  optimize_to_convergence();
}
```

Figure 6-6: A simplified view of the SVM Light code.

to this function until the exit (the interval *inside* the phase-parent) encompasses the region of higher miss rates. The intervals before the entry and after the exit (*outside* the phase-parent) have a significantly lower miss rates than inside.

Figure 6-6 shows a simplified view of the code. The execution intervals outside of `optimize_to_convergence` consist of input output file operations and some preparations before starting the actual learning process. Function `optimize_to_convergence` performs the substantial part of the computation. As a result, the miss rates outside of this function are relatively low compared to inside.

Figure 6-4 shows 470 calls to the phase-marker during the program execution. All these calls are made within the higher miss rates region, because by definition the phase-marker is dominated by the phase-parent, and the call to the phase-parent encompasses the higher miss rates region.

Figure 6-5 shows a zoomed-in view within this higher miss rates region. This figure is similar to Figure 6-2, except that now the entries and exits to/from `optimize_svm`

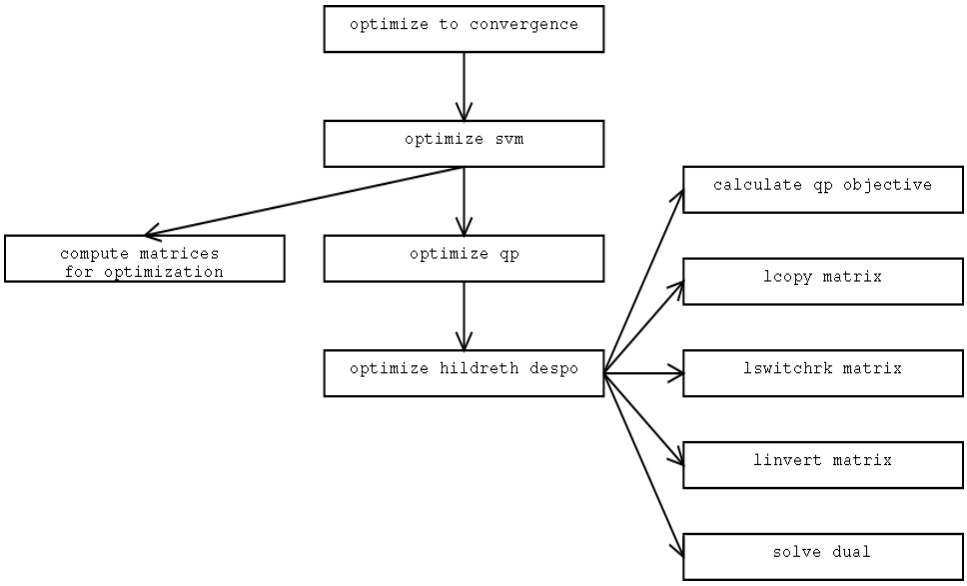


Figure 6-7: A section of the dominator tree under `optimize_to_convergence`, only showing the child `optimize_svm`.

are annotated in the plot. If we take the interval between two consecutive entries to the phase-marker and "stamp" it side-by-side multiple times, we get a similar repeated pattern as is shown in the plot. Likewise, if we take the interval between two consecutive exits, we get a single pattern that was repeated periodically.

Furthermore, the intervals inside `optimize_svm` have zero miss rates, although these intervals occur within `optimize_to_convergence`. As we see before, the intervals inside `optimize_to_convergence` contain much higher miss rates than outside. Thus we can infer that `optimize_svm` by itself has good cache performance, but its parent and "siblings" have bad cache performance.

Figure 6-7 shows the section of the dominator tree rooted in the phase-parent `optimize_to_convergence`, but we omit the children other than `optimize_svm`. Figure 6-8 shows the rest of the tree. We see that the subtree under `optimize_svm` appears to work only on a limited set of data. The children of `optimize_hildreth_despo` all suggest matrix operation procedures, while the other functions appear to prepare the matrix to be "transformed." These names explain why the miss rates inside `optimize_svm` are lower than outside. As the program only works with matrices, as long as the matrix entries are already stored in the cache, the program doesn't need

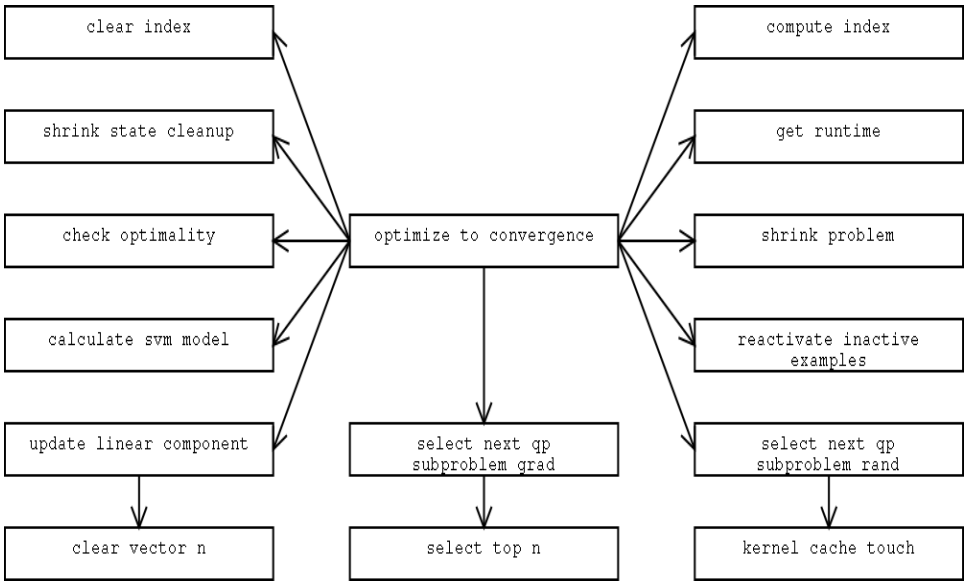


Figure 6-8: A section of the dominator tree under `optimize_to_convergence`, omitting the child `optimize_svm`.

much additional data. Most of the accesses are related to reading and writing the matrix elements, which are already in the cache, except the first time they are brought into the cache. As shown in Figure 6-5, slighly after each entry to `optimize_svm` the miss rates are non-zero for a short period of time, and subsides to zero until the program exits from this function. This period is where some of the matrix entries are being read for the first time and thus the cache suffers occassional miss.

In contrast, Figure 6-8 shows the names of the functions outside of `optimize_svm`. These names suggest that a new matrix is being loaded and computed. In particular, the names of the non-leaves functions, such as `update_linear_component` and `select_next_qp_subproblem_grad/rand`, suggest that these routines take the transformed matrix from `optimize_svm`, use it to update the linear components of the problem, and search the next matrix to be transformed. These names explain why the miss rates in these functions are significantly higher than inside `optimize_svm`. By the time `optimize_svm` is called, the cache are primed with the matrix entries and hence `optimize_svm` exhibits good cache performance.

Figure 6-9 shows a slightly zoomed-in view of Figure 6-5 with entries and exits to both `optimize_svm` and `update_linear_component`. In this figure, the entries are

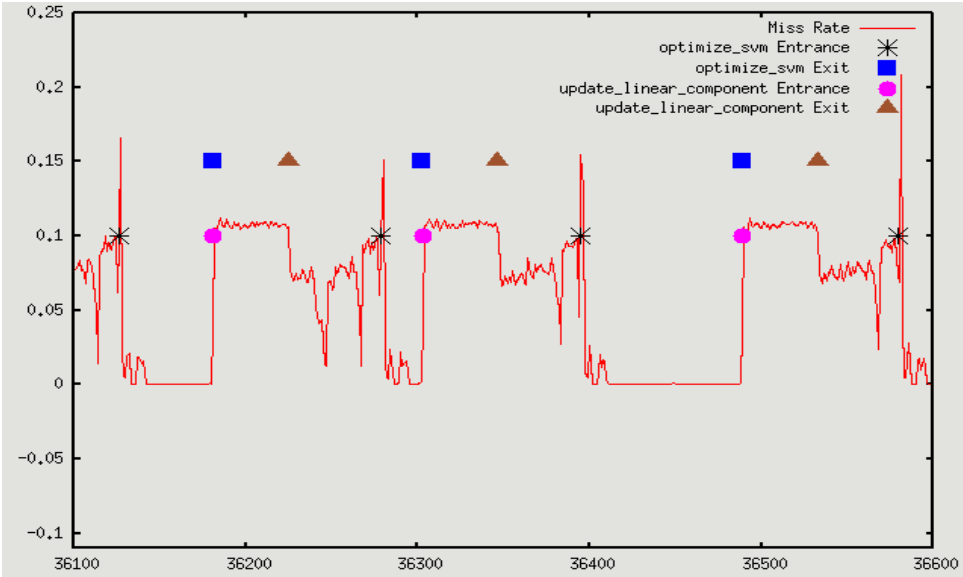


Figure 6-9: A slightly zoomed-in view of Figure 6-5, with the entries and exits to `update_linear_component`.

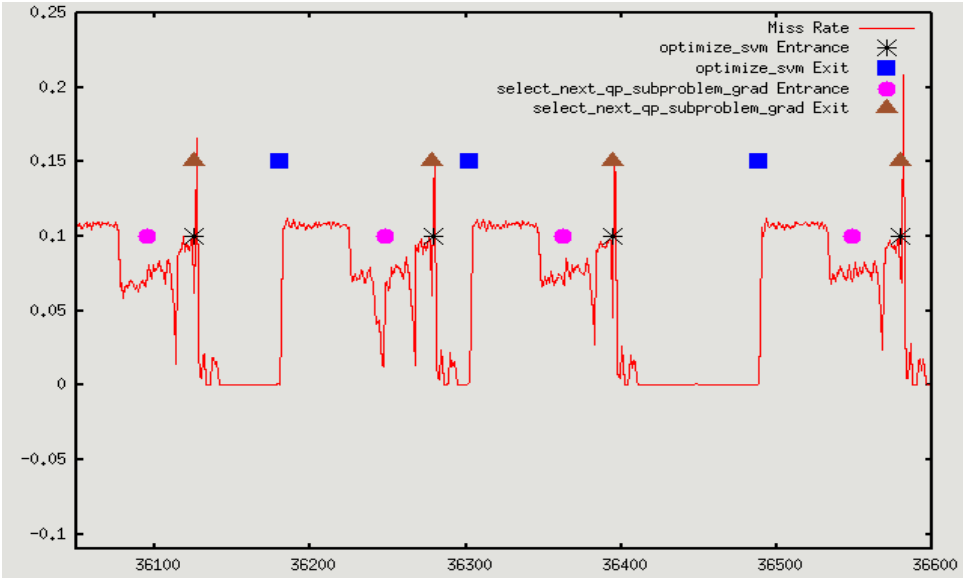


Figure 6-10: A slightly zoomed-in view of Figure 6-5, with the entries and exits to `select_next_qp_subproblem_grad`.

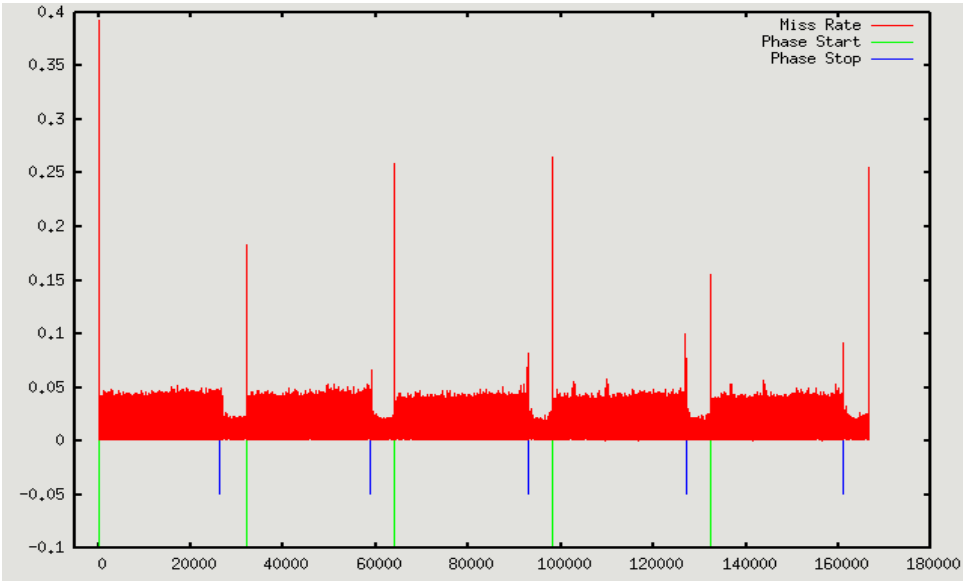


Figure 6-11: Entries to and exits from `spec_compres` in the gzip benchmark.

| Phase-marker | Phase-parent |
|---|---|
| flush_block | deflate |
| spec_compress | main |

Table 6.1: The phase-markers from the GNU Zip Benchmark and their parents.

denoted by symbols at height 0.1 and the exits are denoted by symbols at height 0.15 (see the legend). Likewise, Figure 6-10 shows a similar region, but showing entries and exits to `optimize_svm` and `select_next_qp_subproblem_grad` instead.

We see that these two functions are indeed responsible for the higher miss rates in each of the repeated pattern. Also, these two functions are siblings of `optimize_svm` (they both are immediately dominated by `optimize_to_convergence` as well). Our LogiView does not label them as phase-markers because the subtree underneath these functions are too small. The subtree dominated by `update_linear_component` only contains `clear_vector_n` other than itself. Likewise, the subtree that is dominated by `select_next_qp_subproblem_grad` only contains `select_top_n` other than itself.

### 6.1.2 SPEC Gnu Zip

We run our algorithm for the SPEC 164.gzip Benchmark [36], with window size 8192. The phase-markers and the corresponding phase-parents are shown in Table 6.1. The

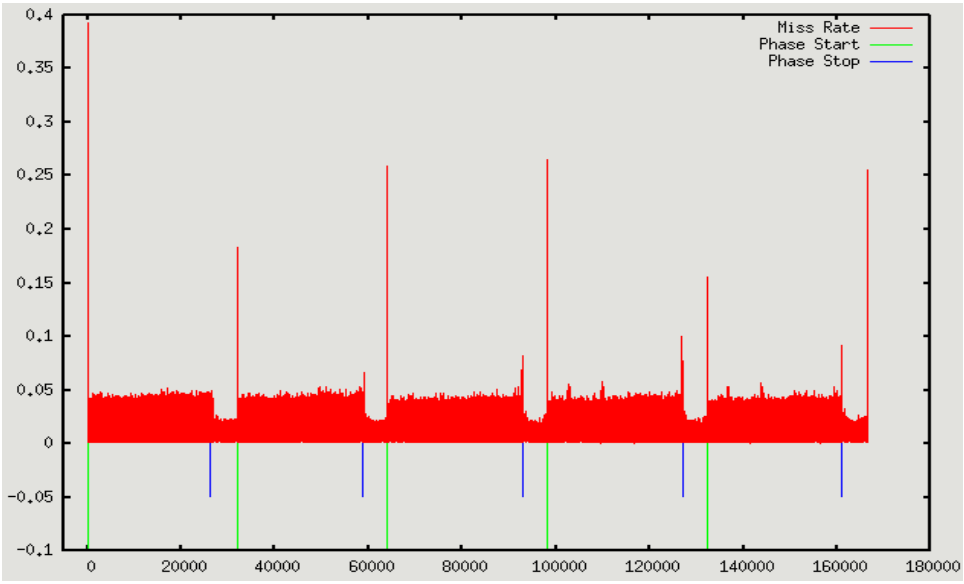


Figure 6-12: Entries and exits to `deflate` in the gzip benchmark.

highest-level phase-marker is `spec_compress` and the corresponding phase-parent is `main`. Obviously, there is only one call to `main` and the call encompasses the entire program execution. Figure 6-11 shows the miss rate plots annotated with the entries and exits from/to `spec_compress`. We see that the interval between two consecutive entries or exits to this function is a pattern that is repeated 5 times in the program. From examining the dominator tree, as discussed in Chapter 3, we know that each of these calls to `spec_compress` compresses and decompresses an input file. Each run of the benchmark tests 5 different files (a large TIFF image, a webserver log, a program binary, random data, and a source tar file ) [36], and the periodic pattern is repeated five times in the program. This evidence strongly suggests that each period of the pattern corresponds to compressing and decompressing one input file.

Another phase-marker in the gzip benchmark is `flush_block`, and the corresponding phase-parent is `deflate`. Figure 6-12 shows the entries and exits from `deflate`. This plot is very similar to Figure 6-11, confirming our previous suspicion that the function `deflate` is where the actual compression and decompression occurs.

Figure 6-12 also demonstrates that the intervals inside `deflate` are considerably different that outside. The intervals inside have a higher miss rates than outside, and the change from high to low or vice versa is very abrupt. We explain this phe-

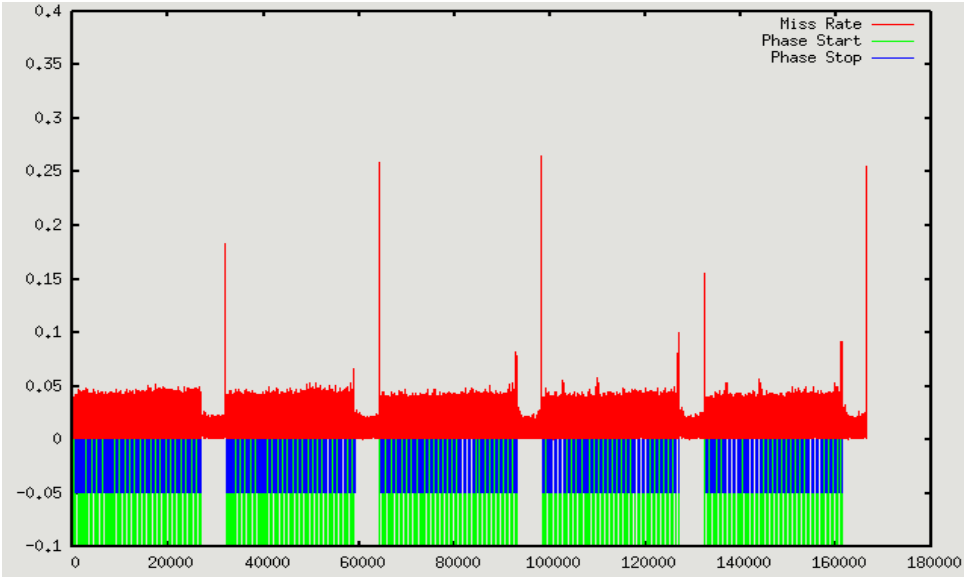


Figure 6-13: Entries to and exits from `flush_block` in the gzip benchmark.

nomenon by arguing that the compression and decompression process reuse data with relatively less frquency. The compression procedure continuously feeds on new plain text data and compresses them. Then the program saves the compressed text back into the memory. The program then decompresses that text to obtain a plain text and compares it against the input file. The decompression procedure does a similar task, which explains why the program experiences a higher miss rates when performing compression and decompression. The intervals outside of `deflate` have lower miss rates because the program does not perform compression and decompression in those intervals. The functions outside of `deflate` mostly deal with opening and loading the input file into the memory. This operation warrants high miss-rates, as we can see in Figure 6-12 that the miss-rates are quite high too. But the miss-rates inside `deflate` are much higher because the operations involve reading the input file and performing compression and decompression on the data.

Figure 6-13 shows the entries to and exits from the phase-marker `flush_block`. Because `flush_block` is dominated by `deflate`, all calls to `flush_block` occur within the calls to `deflate`, and as we have seen, these are the regions of higher miss rates. Figure 6-14 shows a zoomed-in view of Figure 6-13, but this time the entries and exits are shown in circles and triangles respectively. Surprisingly, within the regions

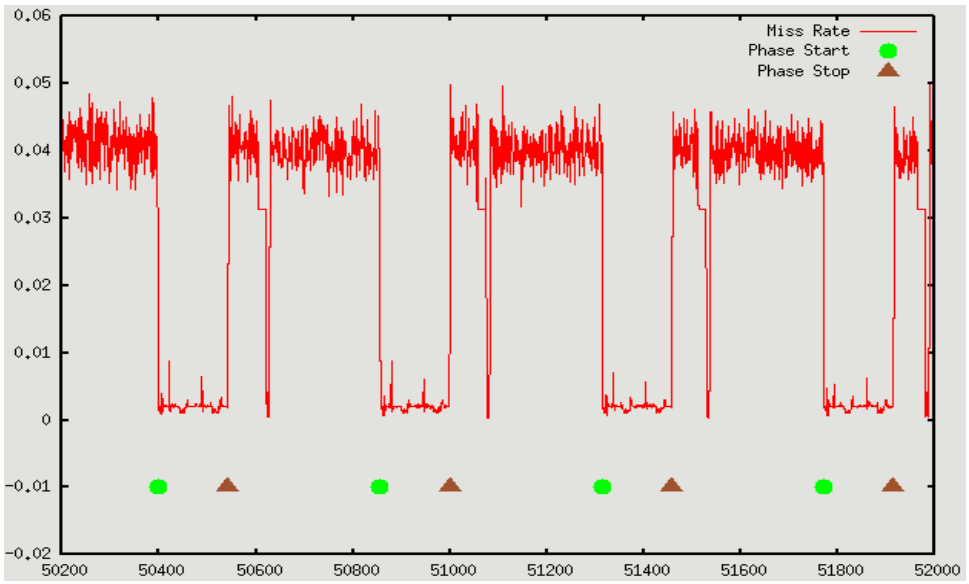


Figure 6-14: A zoomed in view of Figure 6-13, showing a repeated pattern between two consecutive entries or exits from `flush_block`.

of higher miss rates inside `deflate`, there are periodic regions of almost zero miss rates, and these periodic regions occur inside `flush_block`. It also means that the interval between two consecutive entries or exits contains a pattern that is repeated periodically.

### 6.1.3 Timberwolf Benchmark

We run our algorithm for the SPEC 300.twolf Benchmark [38], with window size 32768. Table 4.1 lists the phase-markers and phase-parents found in the program. As we have discussed in Chapter 4 the significance of these pharse-markers and phase-parents, we will examine those functions which mark the logical structure of the program.

First we will examine the phase-marker `uloop` and its parent `utemp`. Figure 6-15 shows the execution of the program, and the entry and exit to `utemp`. We see that the call to `utemp` covers almost the entire execution of the program. In the code, the function `main` almost immediately delegates its task to `utemp`, and thus we can regard `utemp` as the main execution of the program. The intervals outside of `utemp` consist mainly of marshalling the arguments to `utemp`, and thus we can expect these intervals

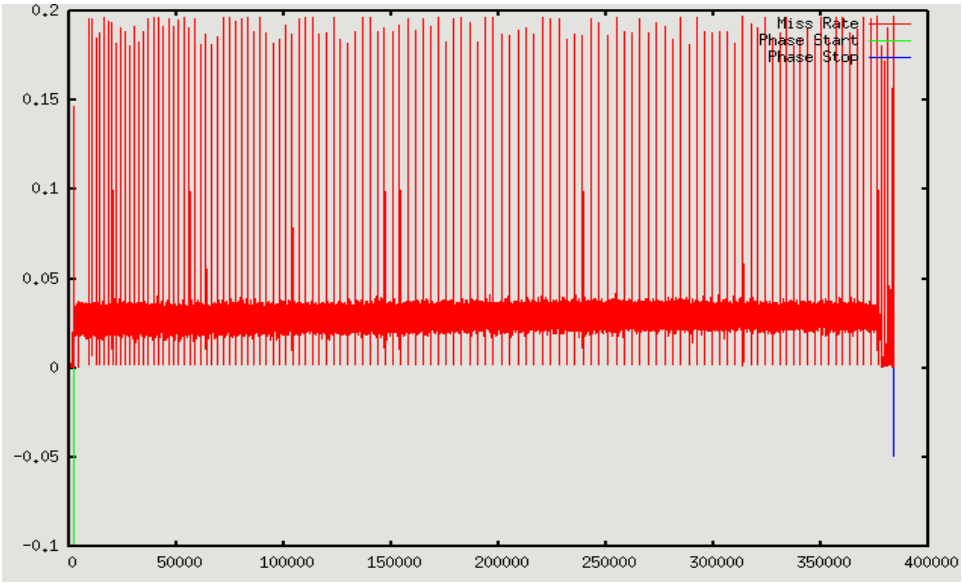


Figure 6-15: Entry and exit from `utemp` in the Timberwolf Benchmark.

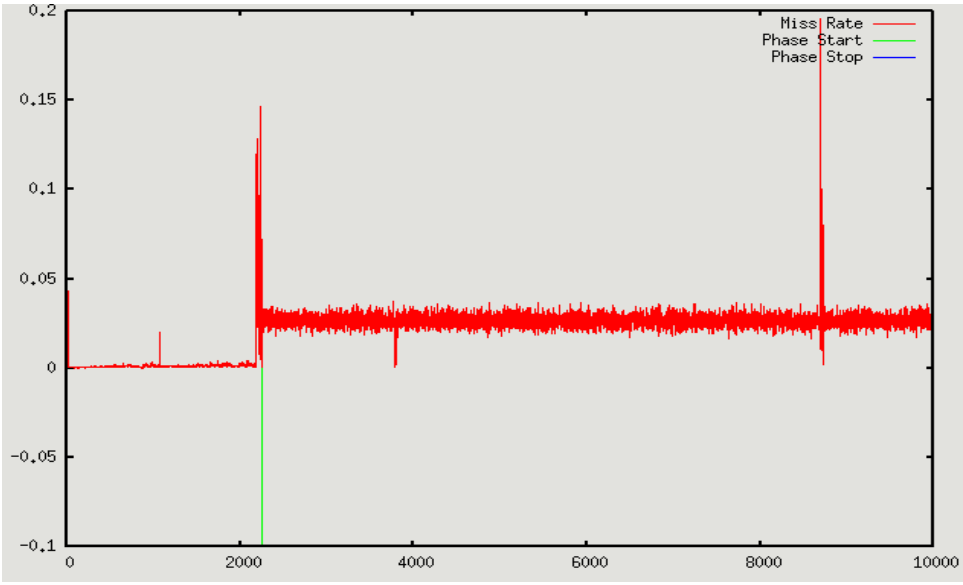


Figure 6-16: A zoomed in view of Figure 6-15, showing the entry to `utemp` in the Timberwolf Benchmark.

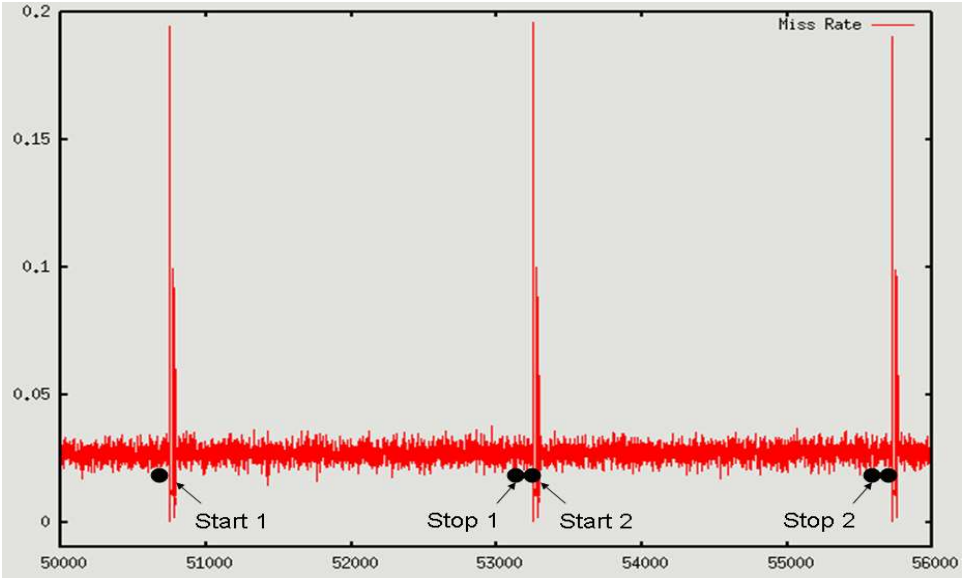


Figure 6-17: Entries and exits from `uloop` in the Timberwolf Benchmark, showing two full repetitions.

to be different that those inside `utemp`. Indeed, figure 6-16 shows the zoomed in view where we only see the entry to `utemp`. The miss rates outside of `utemp` (before the entry to `utemp`) are almost zero where the miss rates inside are considerably higher.

The function `uloop` is called 120 times in the program. Figure 6-17 shows a section of the program execution, showing the entries and exits from `uloop`. Only two repetitions are shown, and the exit from the $n$-th call from `uloop` is immediately followed by the entry to the $(n+1)$-st call. We see that two consecutive entries or two consecutive exits from the function comprises an interval which is repeated throughout the program execution. Slightly after an entry, there is a "spike" (short period of high miss rates), and then the miss rates go back to the stable rate around $0.02 - 0.03$. This behavior repeats multiple times throughout the execution of the program. In our analysis in Chapter 4 we show that `uloop` represents the "inner loop" of the pseudocode that is repeated until there are 20 new states generated (Figure 4-1 line 4). As the program repeats this loop, the behavior of the miss rates is also repeated.

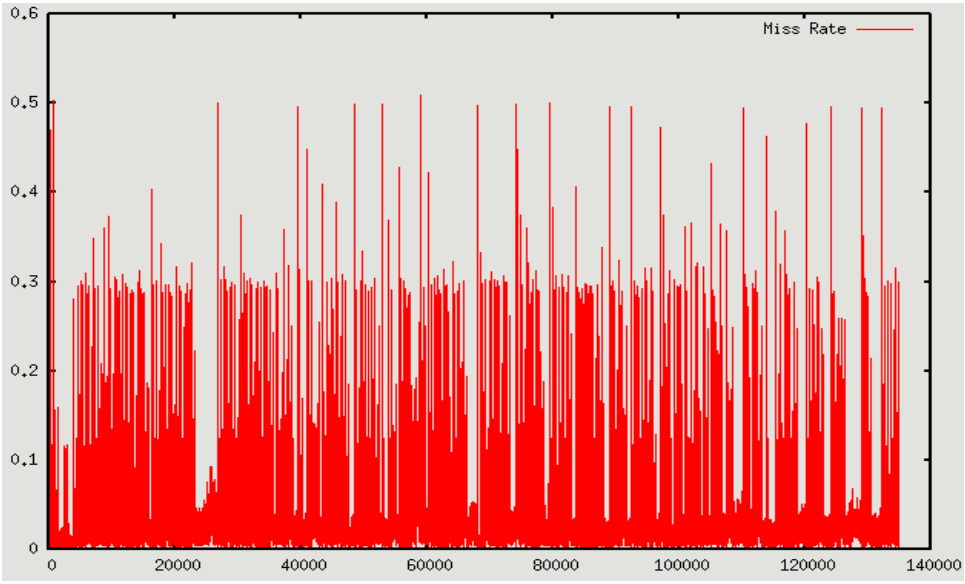


Figure 6-18: The entire execution of the Grammar Link Parser.

### 6.1.4 Grammar Link Parser

We run our algorithm for the SPEC 197.parser Benchmark [37], with window size 32768. Figure 6-18 shows the miss rates over the entire program execution. Table 4.2 lists the phase-markers and phase-parents found in the program. We already discussed that the correlation between these functions are the pseudocode in Figure 4-7. We will see the correlation between these functions and the miss rate.

Figure 6-19 shows the entries to and exits from the phase-marker `read_entry`. We only show two repetitions in this figure. The exit from the $n$-th call is always followed by te entry to the $(n+1)$-st call to this function. We see that the interval between two consecutive entries or exits contains a pattern that is repeated periodically. In `read_dictionary`, the program reads and parses the input dictionary file. This input file is organized in entries [37]. Therefore, the program reads one entry at a time and creates the necessary objects for the representation of the dictionary entry. As the program moves to another entry, this process is repeated. Hence we see that the miss rates behavior is also repeated.

The function `prepare_to_parse` is recognized by LogiView as a phase-parent. Figure 6-20 shows an entry and an exit to this function. The behavior of the miss

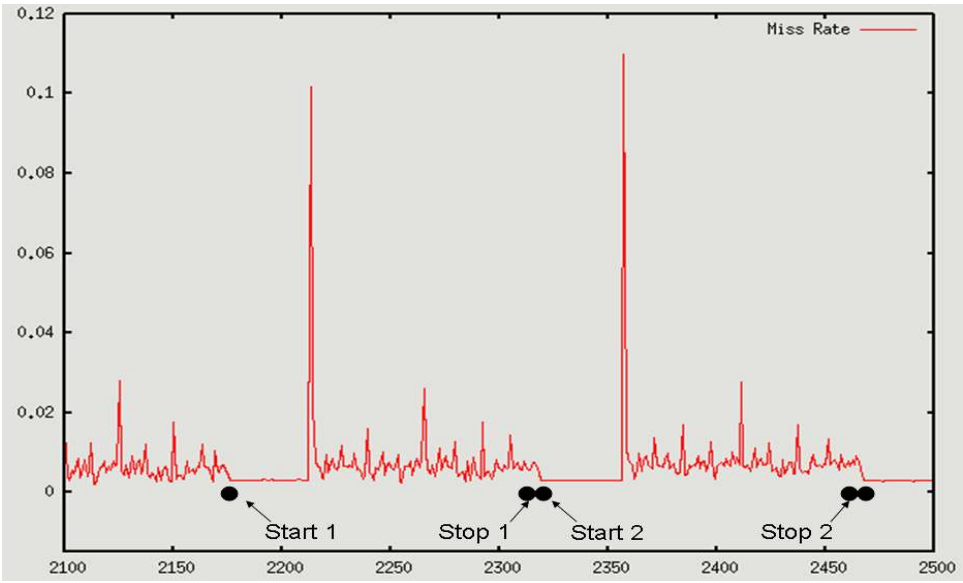


Figure 6-19: Entries and exits to `read_entry` in the Grammar Link Parser.

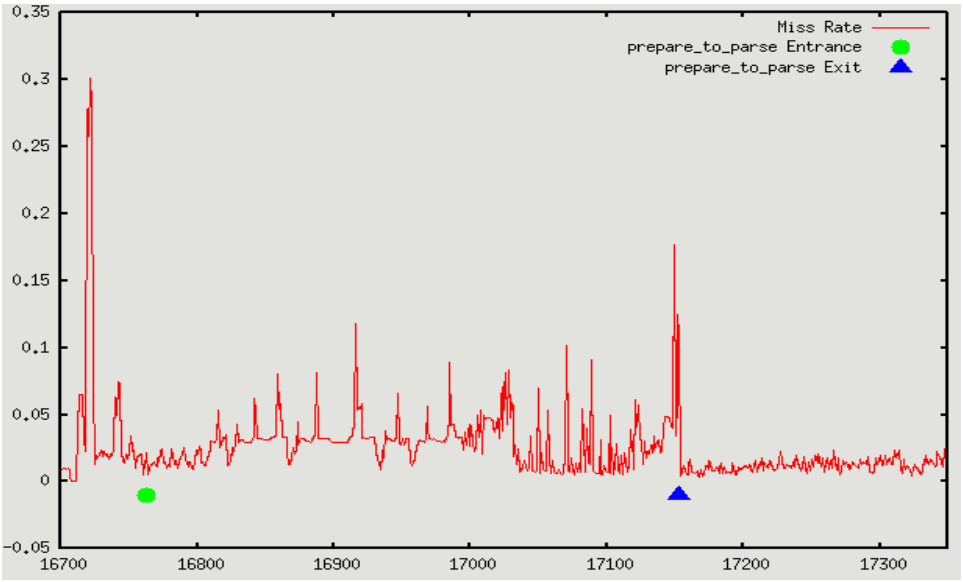


Figure 6-20: Entry and exit to `prepare_to_parse` in the Grammar Link Parser.

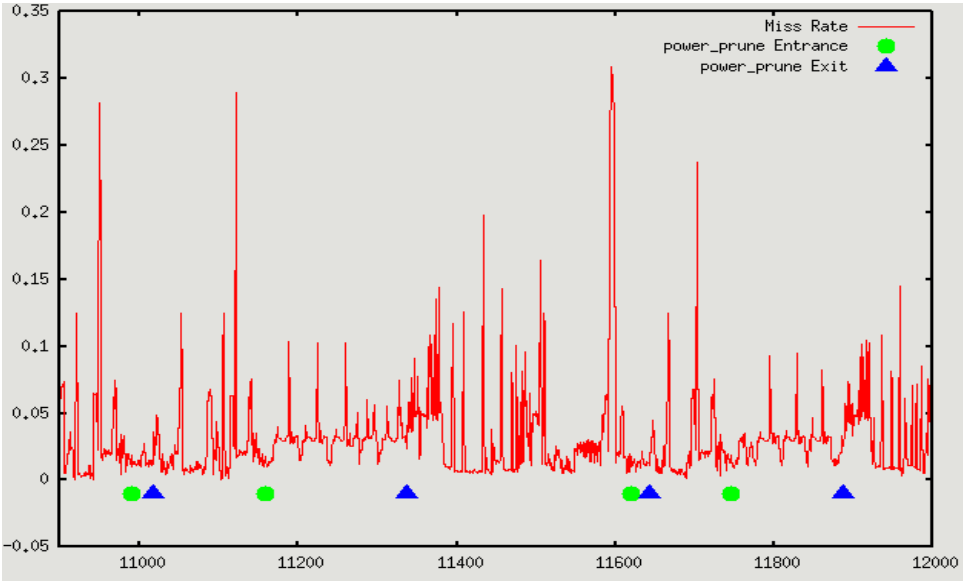


Figure 6-21: Entries and exits to `power_prune` in the Grammar Link Parser.

rates after the exit tends to be stable while the behavior inside the function is much more erratic. The comments in the source code ([37] `main.c` lines 1104-1115,1129-1131) describe that this function does all the necessary pruning and building the grammatical structures. This step takes place after the disjuncts have been generated. After the pruning operations are done and the grammatical structures are built, the program makes one last pass through the sentence in order to match the linkages against the word and output the sentence if necessary. During this one last pass, there are no more pruning operations that occurs. The program simply reads the list of linkages and ensures the consistency of each linkage. As a result, the miss rates are relatively small but nonzero. The rates are also stable because the behavior of the program tends to be uniform.

The corresponding phase-marker is `power_prune`. Figure 6-21 shows the entries and exits of this function. This figure does not exhibit any repeated pattern that is associated with entries and exits to/from `power_prune`. This figure is an example where the phase-marker generated by LogiView fails to exhibit a periodic pattern. The behavior of `power_prune` truly depends on the state of the input sentence. When there are a lot of disjuncts that can be pruned, `power_prune` prunes the disjuncts and adjusts the necessary linkages. As a result, this function makes a lot of data accesses

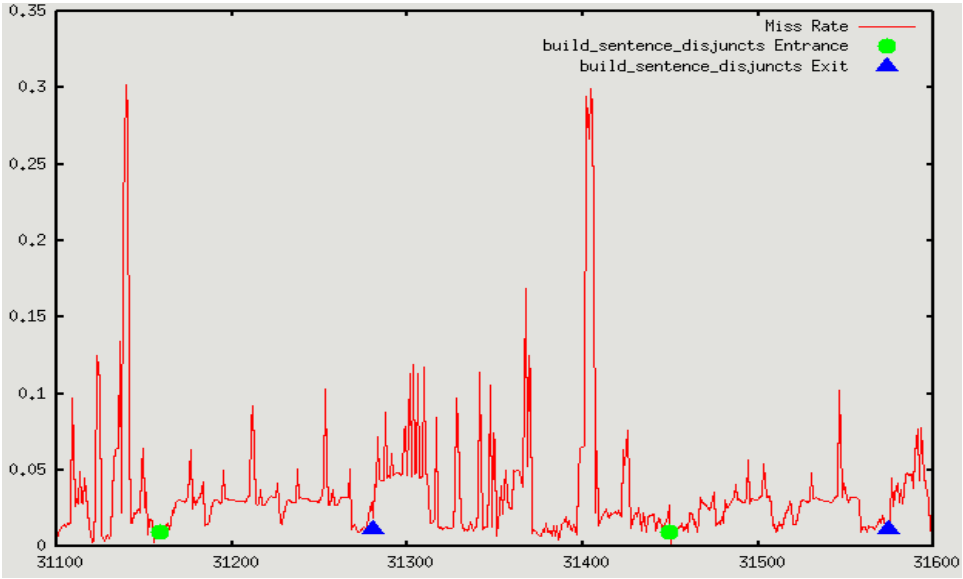


Figure 6-22: Entries and exits to `build_sentence_disjuncts` in the Grammar Link Parser.

in unpredictable places, depending on the previous end of each linkage. Because of this unpredictability, the function fails to exhibit any repeated pattern throughout its multiple calls. In Figure 6-21 we see that there are some resemblances between the second call and the fourth call to `power_prune`. But in general, this pattern is not repeated periodically. This similarity could be because the disjuncts that need to be pruned in those two calls originate from two sentences with similar structures. But when the disjuncts are very different, the behavior ot `power_prune` is also very different.

Figure 6-22 shows the entries and exits to/from `build_sentence_disjuncts`. In Chapter 4 we discussed that this function corresponds to an important step in the program, but it is not recognized by LogiView due to its chain pattern. In Figure 6-22 we see that the function indeed encompasses a repeating pattern, similar to what we would expect from a phase-marker.

Table 6.2 shows the various phase-markers and phase-parents that we have discussed. Most of the time, the phase-parents successfuly identify the phase change and the phase-markers correspond with periodic behaviors. However, there are four exceptions to this trend. Functions `update_linear_component` and `select_next_qp`

| Function name | Recognized by LogiView as | Identifies phase change or repeating pattern |
|---|---|---|
| From SVM Light: | | |
| `optimize_to_convergence` | phase-parent | Yes |
| `optimize_svm` | phase-marker | Yes |
| `update_linear_component` | – | Yes |
| `select_next_qp_subproblem_grad` | – | Yes |
| From GNU Zip: | | |
| `deflate` | phase-parent | Yes |
| `flush_block` | phase-marker | Yes |
| `main` | phase-parent | NA |
| `spec_compress` | phase-marker | Yes |
| From Timberwolf: | | |
| `utemp` | phase-parent | Yes |
| `uloop` | phase-marker | Yes |
| From Grammar Link Parser: | | |
| `read_entry` | phase-marker | Yes |
| `prepare_to_parse` | phase-parent | Yes |
| `power_prune` | phase-marker | No |
| `build_sentence_disjuncts` | – | Yes |

Functions with "–" are not recognized as phase-parent nor phase-marker.

Table 6.2: The phase-markers and phase-parents in identifying phase change or repeating pattern.

`_subproblem_grad` in SVM Light encompass repeating patterns although they are not selected as phase-markers. They do not satisfy the subtree sanity conditions because the subtrees they dominate are too small, only containing one node other than themselves. Similarly, function `build_sentence_disjuncts` in Grammar Link Parser is not recognized as a phase-marker although this function identifies a periodic pattern. Moreover, in Chapter 4 we see that this function also corresponds to a logical step in the algorithm and thus should be marked as a phase-marker. Our LogiView methodology does not mark it as a phase-marker because the subtree it dominates is of a chain pattern. On the other hand, `power_prune` does not exhibit any periodic repetition although it is marked as a phase-marker. Although in Chapter 4 we demonstrate that this function also corresponds to a logical step in the algorithm, the program behavior in a particular call is very dependent on the inputs to that call. As a result, there is no repeated pattern associated with entries and exits to/from `power_prune`. Thus, in Table 6.2, there are 8 functions that behave as expected, 1 false positive and 3 false negatives. We recognized 8 out of 11 phases (recognition rate of 73%) and 8 out 9 phase-markers and phase-parents exhibit behavior change (accuracy rate of 89%)

## 6.2 Correlation to Phase Detection Problem

We have seen that most of the time, the phase-markers and phase-parents accompany a significant behavior change in the program, and the subtree they dominate can explain this change. Although LogiView is primarily designed to help understanding program structure and organization, the phase-markers and phase-parents it produces can also be used for phase detection.

Several works in the past [5, 6, 13] rely on a fixed window size in order to detect phases. Other works [34, 33, 24] use variable window sizes. In our discussion above we use a fixed window size to display the miss rates characteristics. The size for each benchmark is chosen arbitrarily so that the entire execution could fit within one window, and the behavior of each phase is still apparent. However, if the window size

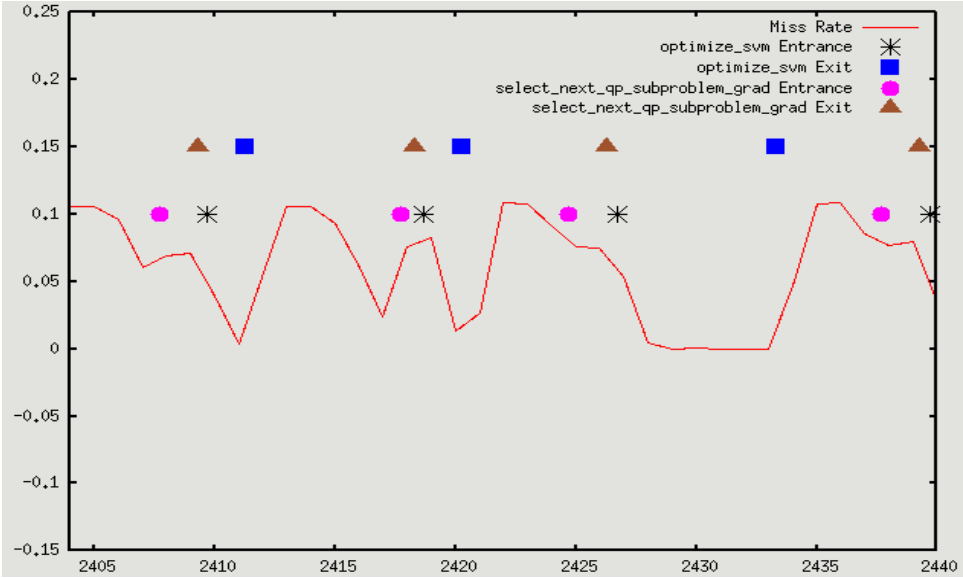


Figure 6-23: Same plot as in Figure 6-10, but with $k = 32768$.

had been chosen incorrectly, we could lose information about the different functions which are responsible for each phase change or phase repetition.

We record the miss rates for the SVM Light Benchmark [15] with window size $k = 32768$ instead of 2048. Figure 6-23 shows the same region as Figure 6-10. We see that the plot becomes very coarse and we lose information by choosing too large a window size. In our discussion in Section 6.1.1, we infer that `optimize_svm` has a good cache performance while `select_next_qp_subproblem_grad` is responsible for the higher miss rate intervals. However, with the large window size and plot as in Figure 6-23, we cannot make such analysis anymore.

Phase detection algorithms that only observe physical behavior need to choose their algorithm parameters carefully. Suppose an algorithm uses a fixed window size and it chooses the window size 32768 for SVM Light. While this algorithm may be able to detect periodicity as shown in Figure 6-10, it may not be able to explain the cause for the repetition or the functions which are responsible for the different regions in each period. However, with our methodology, the window size does not matter in order to recognize the phase-markers. LogiView does not rely on window size when choosing phase-markers, thus `optimize_svm` will still be recognized as a phase-marker regardless of the window size we choose for the phase-detection.

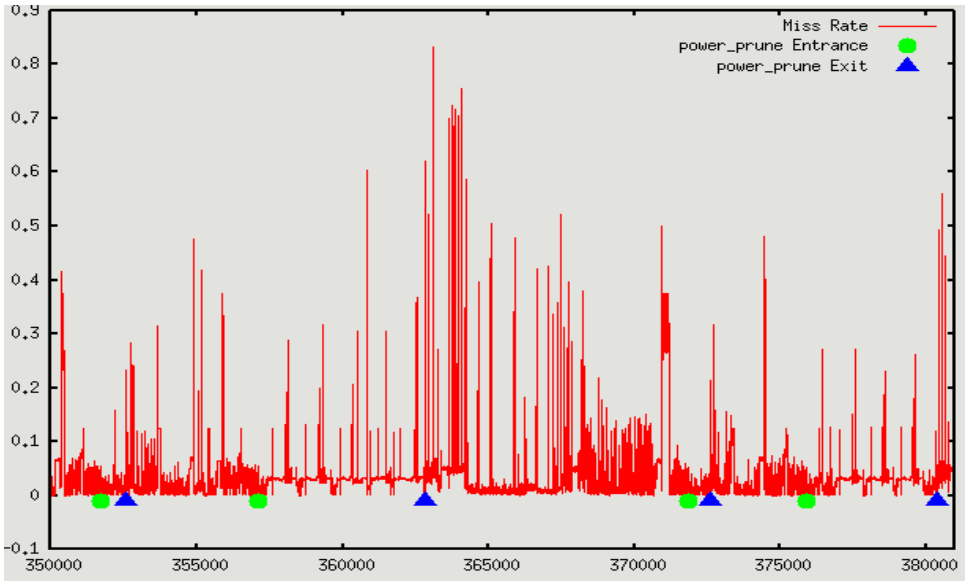


Figure 6-24: Same plot as in Figure 6-21, but with $k = 1024$.

We record the miss rates for the Grammar Link Parser, with window size $k = 1024$ instead of $k = 32768$. Figure 6-24 shows the same region as in Figure 6-21. Function `power_prune` is recognized as a phase-marker, although it does not exhibit any repeating patterns. In our discussion from Chapter 4, we see that `power_prune` is an important step in the program and it corresponds to one of the main instructions in the pseudocode. However, the miss rates behavior for this function is rather unpredictable and display little or no similarity from one call to another. Thus, if a phase detection algorithm only relies on observable physical characteristics, it is unlikely to detect `power_prune` to be a phase in the program. Statistical and numerical analysis of the physical behavior alone will not reveal that `power_prune` is an important step in the algorithm.

# Chapter 7

# Discussion and Conclusion

We presented LogiView, an algorithm to choose several functions in the program as phase-markers. They provide a logical view of the program, and the subtree dominated a phase-parent indicates an organizational structure in the program. The tasks of the function in this subtree are related one to another, and together they constitute a structural part of the program. We demonstrated that this is true for a number of benchmarks. We showed that by examining a phase-parent and the subtree it dominates, we could infer the functionality of the subtree, its role in the program execution and its correlation to the pseudocode.

LogiView separates the functions into disjoint groups, regardless whether they are recognized as phase-markers or not. We leveraged this grouping and combined it with function names analysis to build a dictionary containing the tokens that are most relevant to each application class. We also provided a formula to quantify the similarity of a program to a class. We showed that this formula works well for most applications and most classes as long as the programmers who write the class have a strong naming convention. Therefore, we showed that there are a tremendous amount of semantic information contained in the function names. When viewed in the context of dominator tree, this semantic information helps us understand the functionality of the program.

Lastly, we also showed that our LogiView algorithm can be used as a phase detection algorithm. From the program performance point of view, the phase-markers and

phase-parents accompany the behavior changes and behavior repetitions during the program execution. Furthermore, when viewed in context of the dominator tree, the phase-markers can explain the program performance such as high or low miss rates in certain intervals.

## 7.1 Future Research

There are several possible improvements to our result.

1. The subtree sanity condition has proven wrong several times. There are several functions that do not satisfy this condition but otherwise are good candidates for phase-markers. Further research can formulate this condition better. For example, when does a chain pattern be prohibited from becoming a phase-marker.

2. Our algorithm operates in a two-pass fashion. The first pass runs the program and builds the dominator tree. As the phase-markers are chose, we run the program the second time and signal a phase change upon entering or exiting from the phase-markers and phase-parents. It is possible to build the dominator tree in an on-line fashion. That is, as the program executes, the call graphs are updated and the tree is recalculated if necessary. However, the phase-markers and phase-parents also appear and disappear as the tree evolves its structure. Our difficulty in coming up with an online phase detection primarily concerns the situation when the program entered a new phase under a phase-marker, and then the subtree disappears because a lot of nodes in that subtree were promoted.

3. Our approach for extracting semantic information from function names splits the function names into tokens. In this approach, we lose information that are formed when the right sequence of words are joined together. For example, a function with name `solve_variable_set` would appear no different than a function with name `set_variable_solve`. But those two names actually suggest

a very different semantic meaning. Future work can consider taking the word order into account when analyzing function names.

4. When comparing the tokens from function names, we only look for exact match. So function `solve_matrix`, `solve_matrices`, and `matrix_solver` are all considered different. In reality, we may want to consider them similar or the same. Thus, the names analysis ought to be combined with techniques from Natural Language Understanding such as word root recognition, sentence parsing, etc.

5. In Chapter 6, we explain the high and low miss rates in terms of entries and exits to/from phase-markers and view them in the dominator tree. We understand that the miss rates in an interval are high because the function names suggest so. This analysis could be expanded further to combine our phase detection mechanism and the function names analysis.

# Bibliography

[1] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 156–165, New York, NY, USA, 2005. ACM Press.

[2] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[3] Shantanu Chakrabartty. Gini-support vector machine, a probabilistic support vector machine. Website. http://bach.ece.jhu.edu/pub/gert/svm/ginisvm/.

[4] R. Collobert and S. Bengio. SVMTorch: Support Vector Machines for Large-Scale Regression Problems. *Journal of Machine Learning Research*, 1(2):143–160, 2001.

[5] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working setanalysis. *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 233–244, 2002.

[6] Ashutosh S. Dhodapkar and James E. Smith. Comparing program phase detection techniques. *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 217–227, 2003.

[7] Niklas Een and Niklas Sorensson. An extensible SAT-solver. *Proc. SAT*, 3, 2003.

[8] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[9] Zhaohui Fu. zchaff. Website. http://www.princeton.edu/c̃haff/zchaff.html.

[10] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[11] Murali Haran, Alan Karr, Alessandro Orso, Adam Porter, and Ashish Sanil. Applying classification techniques to remotely-collected program execution data. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 146–155, New York, NY, USA, 2005. ACM Press.

[12] M.J.H. Heule and H. van Maaren. March dl: Adding Adaptive Heuristics and a New Branching Strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:47–59, 2006.

[13] Michael J. Hind, Vadakkedathu T. Rajan, and Peter F. Sweeney. Phase shift detection: A problem classification. Technical Report RC23058, IBM Research, 2003.

[14] Vladimir Jakobac, Alexander Egyed, and Nenad Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In *Lecture Notes in Computer Science*, volume 3442, pages 253–268. Springer Berlin / Heidelberg, January 2005.

[15] T. Joachims. Making large-scale svm learning practical. In B. Scholkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT-Press, 1999.

[16] K. Lang. Evidence-Driven State Merging with Search. *NEC Research Institute*, 1998.

[17] Kevin J. Lang and Barak A. Pearlmutter. Abbadingo one: Dfa learning competition. Website. http://abbadingo.cs.unm.edu/.

[18] K.J. Lang, B.A. Pearlmutter, and R.A. Price. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. *Lecture Notes in Computer Science*, 1433(1):12, 1998.

[19] James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, New York, NY, USA, 1999. ACM Press.

[20] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 135–146, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[22] Bill McCloskey and Eric Brewer. Astec: a new approach to refactoring c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[23] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117, 1995.

[24] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance data collection using a hybrid approach. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 126–135, New York, NY, USA, 2005. ACM Press.

[25] H. Mittelmann. Benchmark of free linear programming solvers. Website. http://plato.asu.edu/ftp/lpfree.html.

[26] Steven Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[27] Shashidhar Mysore, Banit Agrawal, Timothy Sherwood, Nisheeth Shrivastava, and Subhash Suri. Profiling over adaptive ranges. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 147–158, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability. Master's thesis, Tel-Aviv University, 2002.

[29] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2 and 3):103–116, 1997.

[30] R.M. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report 646, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, June 2004.

[31] SATcompetition. Sat competitions. Website. http://www.satcompetition.org/.

[32] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *Solid-State Circuits, IEEE Journal of*, 20(2):510–522, 1985.

[33] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural*

*support for programming languages and operating systems*, pages 165–176, New York, NY, USA, 2004. ACM Press.

[34] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *Micro, IEEE*, 23(6):84–93, 2003.

[35] Alex Smola and Bernhard Scholkopf. Kernel machines repository of machine learning algorithms. Website. http://www.kernel-machines.org/.

[36] SPEC Benchmark. 164.gzip: SPEC CPU2000 benchmark description. Website. http://www.spec.org/cpu/CINT2000/164.gzip/docs/164.gzip.html.

[37] SPEC Benchmark. 197.parser: SPEC CPU2000 benchmark description. Website. http://www.spec.org/cpu/CINT2000/197.parser/docs/197.parser.html.

[38] SPEC Benchmark. 300.twolf: SPEC CPU2000 benchmark description. Website. http://www.spec.org/cpu/CINT2000/300.twolf/docs/300.twolf.html.

[39] SPEC Benchmarks. Standard Performance Evaluation Corporation. Website. http://www.spec.org/cpu/CINT2000/.

[40] Kavitha Srinivas and Harini Srinivasan. Summarizing application performance from a components perspective. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 136–145, New York, NY, USA, 2005. ACM Press.

[41] Ofer Strichman and Roman Greshman. Haifasat - a new robust sat solver. Website. http://www.cs.technion.ac.il/ gershman/HaifaSat.htm.

[42] Davy Temperley, Daniel Sleator, and John Lafferty. Link parser api. Website. http://www.link.cs.cmu.edu/link/api/index.html.

[43] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on*

*Foundations of software engineering*, pages 197–206, New York, NY, USA, 2005. ACM Press.

[44] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.