Abstract

VANKA, RAJESHWAR. Efficient Data Dependence Profiling. (Under the direction of James Tuck.)

With processor frequencies leveling off and core designs remaining relatively static, optimizing existing software to achieve higher performance is increasingly important. Prior works have shown that speculating on conservative data dependence constraints can unlock performance potential within existing codes. Techniques such as thread-level speculation and speculative optimizations leverage this property to perform risky transformations to boost performance. In order to maximize the potential of these techniques, accurate knowledge of the data dependences in question is necessary. Data dependence profilers (DDPs) provide this information by tracking dependences during execution using representative inputs.

In this thesis, I'll first describe and evaluate a conventional data dependence profiler, and expose its shortcomings. Traditional profilers use a centralized structure to track dependences. This leads to slow and inaccurate profiling. Subsequently, I’ll introduce a novel set-based profiling scheme which uses sets to provide decentralized storage and operations. The scheme utilizes static analysis to precisely identify the dependences that need to be tracked, and performs a greedy set allocation to group the instructions (and hence dependences) across sets.

Software signatures (simple bloom filters) are used to realize the sets at runtime. The signatures are highly tunable, and provide quick, albeit imprecise, operations. Using these operations, we are able to track dependences at runtime. With the help of smart optimizations, the scheme can profile large programs quickly and accurately. Significantly, tests show that the performance (slowdown versus uninstrumented code) of the profiler can be significantly improved without necessarily sacrificing accuracy.

The profiler is evaluated using queries from a Speculative Code Motion pass. The pass identifies scheduling opportunities involving loads and prior stores which might share the same address. These queries cover (on average) > 80% of the memory operations in the SPEC2000 suite of programs. The profiler is able to collect the necessary information with an overhead of 100%, with an accuracy of 0.19 NAED (Normalized Average Euclidean Distance). This compares with an accuracy of 0.40 NAED achieved by a traditional DDP with larger overheads. The profiler achieves this accuracy with 100% overhead versus an aggressively optimized uninstrumented baseline.

This runtime information is then fed back into a state-of-the-art compiler for Feedback Directed Optimizations (FDOs). Combined with edge profile data, traditional optimizations perform risky optimizations to unlock performance within existing codes. The performance is evaluated on several processor configurations using a cycle accurate x86 simulator. Both the
profile data, and its effect on these optimizations are analyzed. We show that there is untapped performance within existing programs that can realized through speculative optimizations, and that data dependence profiling can deliver performance improvements. Using the set-based profiler scheme, a 4.4% improvement in dynamic instruction count is realized, leading to a 5% improvement in execution time. In equake, we see a 20% improvement in instruction count, leading to a >10% improvement in cycles across all configurations. Further experiments showed that these results closely match the opportunity found using an ideal profiler. This shows that the set-based profiler is accurate enough to realize almost all of the opportunity available.

To summarize, this thesis shows that a set-based data dependence profiler can produce quick and accurate results on large programs. We show that our profiler is faster and more accurate than prior approaches. Our set-based allocation schemes are applicable to prior designs of profilers, and can help improve their accuracy. The information collected by the set-based profiler shows performance improvements across the multiple benchmarks using a feedback-directed optimization framework. We show that when combined with edge profile information, the performance of benchmarks can be improved by 5% on average, with a 10% improvement in the best case.
Efficient Data Dependence Profiling

by
Rajeshwar Vanka

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2013

APPROVED BY:

__________________________  _________________________
Xiaosong Ma                  Eric Rotenberg

__________________________  _________________________
Huiyang Zhou                James Tuck
Chair of Advisory Committee
Dedication

To my family.
Biography

The author was born in Hyderabad, the capital of the state of Andhra Pradesh situated on the Deccan Plateau in south India. His parents, Dr. V. S. S. Sastry and Dr. N. Satyavathi, are natives of Andhra Pradesh as well. Dr. Sastry works as a physics professor in University of Hyderabad, whereas his mom, Dr. Satyavathi, worked as a physics professor in Osmania University. His brother, Dr. Sundaram Vanka, works in field of wireless networks and recently completed his PhD from the University of Notre Dame.

The author completed his bachelor’s degree from ICFAI Tech Hyderabad, affiliated to ICFAI University Dehradun. Subsequently, he joined North Carolina State University as a Masters student in Electrical and Computer Engineering. He has been working with Dr. James Tuck for the past five and a half years. In 2009, he received a Masters of Science degree in Computer Engineering from the university. Subsequently, he joined the PhD program in the same department.
Acknowledgements

I come from an academic family, with both my parents working as physics professors. My brother recently completed a PhD in wireless networks. This background provided an excellent foundation for pursuing graduate studies and proceeding towards research. I'm very grateful to my family for providing me with the analytical skills and nurturing interest in the field of my choice. I was inspired to pursue this degree after witnessing their understanding and excitement with their respective areas.

Upon completion of my bachelors degree, I traveled to the US and was met by Sridhar Kudumala and his wife, Kavita. They helped me make the transition and set myself up in Raleigh. I would like to thank them for their help and hospitality.

I met with Dr. James Tuck prior to the first day of classes, and have been interacting with him ever since. He has been the single biggest influence in my career over the past few years, and I'm very much grateful to him for his advise, patience and resolve to ensure that I understood the ins and outs of research, and to place knowledge of the subject above all other considerations. My background in compilers was weak, at best, when I started out at NCSU. I'm deeply grateful for his mentoring, and for showing how to understand and navigate the field. Besides my family, he has been the single biggest influence on my career, and I'm pleased to continue working with him after the completion of the degree.

I would like to thank the thesis committee for their insight comments and suggestions. I have interacted with Dr. Huiyang Zhou and Dr. Xiaosong Ma when I took their courses on GPUs and distributed systems. I believe that understanding a field requires breadth. Their courses, and more importantly, their presence on the thesis committee has been very helpful in this regard. My association with Dr. Rotenberg goes back to Fall 2008, when I was assigned as a teaching assistant for his course on computer architecture. Subsequently, I took his course on advanced microarchitecture, and worked as a teaching assistant for that course. During these times, I have had the chance to interact with him and his group. I admire the clarity in his arguments, and his attention to detail which often exposes interesting observations. It was a pleasure working with him, as my understanding of the subject matter got better each time.

During the course of my PhD, I got the opportunity to work with Dr. Dan Quinlan’s ROSE compiler group in Lawrence Livermore National Laboratory, Livermore, California. I worked for about nine months in the group. During this period, I met Dr. Greg Bronevetsky, Dr. Peter Pirkelbauer, Dr. Chinhua Liao and Justin Too, besides Dr. Quinlan himself. The experience in the lab was a memorable one, teaching me how to work with a large code base, and a group of developers. I would like to thank Dr. Quinlan for providing me the opportunity to work in his group, and Dr. Bronevetsky, Dr. Pirkelbauer, Dr. Liao and Justin for their help in making my
stay at the lab memorable.

Finally, I would like to thank the friends that have made my time at NC State memorable. I would like to thank my roommates and good friends, Sankararaman Janakiraman, Rajeev Patwari, Rajsekhar Santhapoor and Salil Wadhavkar, with whom I shared many good times. The chilly winter vacations, and the deserted summers were more fun thanks to all the hangouts and potlucks.

The CESR lab was a vibrant place, and I made many good friends over the course of my stay. In alphabetical order, they are Siddhartha Chhabra, Niket Choudhry, Sabina Grover, Saurabh Gupta, Sandeep Navada, Poulomi Pal, Salil Pant and Devesh Tiwari. Devesh’s name comes at the end of the list, but he was the first person I met in the lab, and has been a very good friend ever since. I’m very happy that I had the opportunity to share fruitful discussions, and late night chats with them.

My internship at Dr. Quinlan’s group was memorable, and part of the credit goes to the other interns and my relatives in California who made me feel at home. My uncle, Satya Nishtala and his wife, Seetha are my second family now. Their son, Rajesh Nishtala and his wife, Rakhee provided the support necessary to enjoy the time. At the lab itself, I made quite a few friends, and I cherish them even today. In alphabetical order, they are Sriram Ananthakrishnan, Akshatha Bhat, Pei-Hung Lin, Hongyi Ma, Steena Monteiro and Tristan Vanderbruggen. Interestingly, Akshatha and myself joined the same company, and share many stories from the internship times.

Finally, there are a few friends whom I met informally, and the friendships have blossomed over the years. I would like to thank Santosh Navada and Shivam Priyadarshi for their support and encouragement when the times were tough. Halley Tej, Arjun Bangre and Karteek Peri were the first few friends I made at NC State, and I owe them a great deal for helping me settle down in this place. I cherish their friendship, and would like to acknowledge their role in achieving this milestone.

Finally, I would like to especially thank Aishwarya Ramaswamy who has been a friend for more than a decade now, and has been a constant source of humor, laughter and a dose of reality. She is a constant reminder of the life outside the world of engineering.

I would like to thank everyone else, whom I have met during the course of my stay here, and helped me at various stages during this degree.
# Table of Contents

List of Tables ........................................................................... ix

List of Figures .......................................................................... x

**Chapter 1 Introduction** ........................................................... 1
  1.1 Data Dependence Profiling ................................................. 1
  1.2 Traditional Data Dependence Profilers ......................... 2
    1.2.1 Operation ................................................................. 2
  1.3 Set-based Profiling using Software Signatures ............... 3
    1.3.1 Signatures ................................................................. 4
  1.4 Feedback Directed Optimizations ................................. 4

**Chapter 2 Background** ......................................................... 5
  2.1 Alias analysis - data dependences ................................. 5
  2.2 Bloom filters and Software Signatures ......................... 9
  2.3 Signature Operations ...................................................... 10

**Chapter 3 Methodology** ....................................................... 13
  3.1 Host System and Benchmarks ........................................ 13
  3.2 Compiler ................................................................. 13
    3.2.1 Data Dependence Profilers ................................. 15
    3.2.2 Feedback Directed Optimization .......................... 15
  3.3 Speculative Code Motion ............................................. 18
  3.4 Sampling ................................................................. 19
  3.5 Performance and Accuracy Metrics ............................. 19

**Chapter 4 Traditional Data Dependence Profilers** ............. 21
  4.1 Design ................................................................. 21
    4.1.1 Original Design ..................................................... 21
    4.1.2 Streamlined Design ............................................. 22
    4.1.3 Improved Original Design .................................. 23
  4.2 Evaluation ............................................................... 23
    4.2.1 Streamlined Design ............................................. 23
    4.2.2 Improved Original Design .................................. 26
  4.3 Drawbacks ............................................................... 31
    4.3.1 Original Design ..................................................... 31
    4.3.2 Streamlined Design ............................................. 31
    4.3.3 Improved Original Design .................................. 32

**Chapter 5 Set-based Profiling using Software Signatures** .... 33
  5.1 Main Idea ............................................................... 33
  5.2 Set-Based Profiler Design .......................................... 34
    5.2.1 Interference Graph Generation .............................. 35
Chapter 8 Related Work .............................................. 107
  8.1 Data Dependence Profiling ..................................... 107
  8.2 Speculative Optimizations ..................................... 108

Chapter 9 Conclusion .................................................. 110
  9.1 Data Dependence Profiling ..................................... 110
  9.2 Feedback Directed Optimization ............................... 111

References ............................................................... 113

Appendix ................................................................. 117
  Appendix A Tool Description ....................................... 118
    A.1 Profiler ....................................................... 119
    A.2 Feedback Directed Optimizations ........................... 120
List of Tables

Table 3.1 Processor Configurations ........................................... 14
Table 3.2 Profile Coverage - Loads and Stores profiled .................... 18
Table 4.1 Changes in design to streamline the profiler ...................... 23
Table 5.1 Profiler Configurations ............................................... 42
Table 5.2 Accuracy and its impact .............................................. 45
Table 6.1 Characterization of Baseline Profiler. ............................... 60
Table 6.2 System Configurations for Early Termination (ET). ............... 61
Table 6.3 Effect of Early Termination on Conflict Checks ................... 63
Table 6.4 Population Distribution Percentage ................................. 65
Table 6.5 Set Customization Types ............................................ 65
Table 6.6 System Configurations for Set Customization. ..................... 66
Table 6.7 Percentage Increase in Sets of Each Type ......................... 68
Table 6.8 System Configurations for new Population-Based Set Allocation. 69
Table 6.9 Original Set Distribution ............................................ 71
Table 6.10 AllocS Set Distribution .............................................. 72
Table 6.11 Percentage change in Set Distribution ............................. 72
Table 7.1 Processor Configurations ............................................ 91
Table 7.2 Evaluated Configurations ............................................ 93
List of Figures

Figure 1.1 An example of data dependence profiling. .................................. 3
Figure 2.1 An example CFG for a function. .............................................. 6
Figure 2.2 Responses from Alias Analysis .............................................. 7
Figure 2.3 Runtime information gathered by a data dependence profiler ...... 7
Figure 2.4 Profile Data classification and transformation .......................... 8
Figure 3.1 Typical workflow in LLVM .................................................... 14
Figure 3.2 Workflow with Data Dependence Profile ............................... 14
Figure 3.3 Workflow with Edge Profile feedback ..................................... 15
Figure 3.4 Workflow with DDP and Edge Profile feedback to FDO .......... 16
Figure 3.5 Examples of dependence checks for speculative optimizations. 17
Figure 3.6 Figure showing variation of NAED in different scenarios .......... 20
Figure 4.1 Impact of size on Streamlined Hash Tables ............................. 24
Figure 4.2 Impact of sampling on Streamlined Hash Tables with 50000 entries 26
Figure 4.3 Impact of sampling on Streamlined Hash Tables with 100000 entries 27
Figure 4.4 Impact of sampling on Streamlined Hash Tables with 200000 entries 28
Figure 4.5 Tagged Hash Tables with 20 Tags per entry ............................ 29
Figure 4.6 Tagged Hash Tables with 10 Tags per entry ............................ 30
Figure 5.1 Examples of dependence checks for speculative optimizations. 34
Figure 5.2 Stages in Set-based Profiler .................................................. 35
Figure 5.3 Interference Graph using proposed set allocation schemes. ....... 37
Figure 5.4 Allocating QSets to PSets using Interference Graph ............... 39
Figure 5.5 Instrumentation using Disambiguation ................................... 41
Figure 5.6 Instrumentation using Membership ....................................... 41
Figure 5.7 Performance and accuracy comparison of best signature and best hash table designs. ......................................................... 43
Figure 5.8 Set Characterization Part 1 ..................................................... 46
Figure 5.9 Set Characterization Part 2 ..................................................... 47
Figure 5.10 Performance and accuracy comparison of signature designs .... 50
Figure 5.11 Performance and accuracy variation for S_RM_2K using different sample rates ................................................................. 52
Figure 5.12 Performance and accuracy comparison of single array signature designs 54
Figure 5.13 Impact on SpecLCM - Percent of Insts and Loads Moved ....... 56
Figure 5.14 HW Cache Hit Percentage .................................................... 57
Figure 6.1 Early Termination ................................................................. 62
Figure 6.2 Early Termination + Set Customization ................................... 67
Figure 6.3 Early Termination + Set Allocation Using Static Insertion Counting 70
Figure 6.4 Percentage of PSets and Conflict Checks compared to Baseline .... 71
Chapter 1

Introduction

1.1 Data Dependence Profiling

Programs are written such that they can handle a multitude of inputs, that exercise a large number of control and data flow paths within the code. Studies have shown that certain paths (control and data) are exercised far more than others. These paths are commonly called *hot paths*. Compilers which translate these programs into efficient, semantically correct machine code have to account for all the paths within the code, and hence generate code that is frequently sub-optimal in the common case. Speculative optimizations aggressively transform the code by performing risky transformations which are only valid under certain inputs. By optimizing the code for the common case, the resulting code runs faster, most of the time. Monitoring and recovery support ensures that the semantics of the program are never violated. These optimizations can work better if they are provided with runtime information regarding the behavior of the code. In order to continue delivering performance improvements in static and dynamic compilation environments, gathering runtime information quickly and accurately is becoming increasingly important. Profiling is a way of gathering such information.

Edge profiling determines the probability that a given control flow edge (an edge connecting two basic blocks in a program) is taken. This information helps the compiler bias its decisions towards the more frequently taken path. Data dependence profiling determines the probability that a given data flow edge (an edge connecting two memory operations) occurs. This allows the compiler to optimize the code surrounding the instructions better. As an example, an invariant load that doesn’t have a data flow edge to any other stores in a loop, can be hoisted from the loop. In addition to the load, its slice is now eligible for hoisting as well.

Edge profiling can be performed with low overhead, and has been leveraged in hyperblocking and other trace-based techniques. Data dependence profiling is expensive as it needs to collect and process much more information at runtime compared to edge profiling. Prior works
on VLIW processors and speculative loop parallelization techniques have shown that data dependence profiling can help unlock performance hidden within existing codes. Therefore, in this thesis, we describe a new set-based data dependence profiling technique that is both faster and more accurate than prior works.

The need for data dependence profiling arises from the transformations that optimizations try to perform, but are unable to, due to conservative alias analyses. Prior works have designed data dependence profilers that can collect runtime information for entire programs. However, in our studies, we show that these profilers are either too slow, inaccurate or cannot capture certain relationships within the program at all. We describe the design of these profilers in the next section. The reader may find a more detailed study of these profilers in Chapter 4.

1.2 Traditional Data Dependence Profilers

As mentioned previously, prior works [9, 12, 24] have proposed data dependence profilers. These profilers have many similarities, hence we refer to them as traditional data dependence profilers. We base our discussion here on [9], which best describes their design.

These profilers use a centralized structure, typically with functionality similar to that of a hash table, to record the dependence information for a given region. In most cases, the region refers to the entire program, however it can be tailored to work with functions or loops. The structure(s) is(are) designed such that instructions accessing the same location are recorded as dependence chains. In a typical design, consecutive entries in the dependence chain refer to consecutive instructions that access the same location in memory. In order to form these dependence chains at runtime, all instructions referring to a given memory location share an entry in the structure. Each time an instruction needs to access the structure, it uses the address (its accessing in memory) to lookup the entry in the table. Hence, in order to form the dependence chains accurately, the size of the structure should be equal to number of unique addresses accessed by the program. This is an exorbitant cost, and clearly not scalable to large programs. For this reason, most techniques restrict the size of the table, which paves the way for false dependences to creep into the chains.

1.2.1 Operation

The profiler tries to form chains by linking each instruction with the closest prior instruction that shares the same location. Figure 1.1 describes the operation of the profiler. Part(a) shows the uninstrumented code. Each instruction is assigned a unique RefID (Reference ID), which will be used to concisely represent the instruction and for feedback. Part(b) shows the annotations. The annotations call libraries which manage the global data structures used by the profiler.
Such structures may be fixed-size tables or large and virtually unlimited hash tables. In most medium-to-large programs, a fixed-size table is necessary as infinite size hash tables quickly consume all available memory. Parts (c) and (d) show an insertion operation. At load Y, the table is indexed using the address (Y), which clashes with an entry for address B (entered at store B) in the table. A dependence between 9 (current instruction) and 2 (prior instruction) is recorded. Instruction 9 then replaces 2 as the most recent instruction to that index.

Given a fixed-size table design, like Part(d), it is possible that B and A share the same entry in the hash table. In this case, a false dependence between instructions 9 and 1 would be recorded. Even with proper calculation of the dependence chains, this design does not provide accurate answers required by optimizations, to speculatively transform the code. This profiler design is described in greater detail, and evaluated in Chapter 4.

1.3 Set-based Profiling using Software Signatures

In order to overcome the problems with existing DDPs, we took a fundamentally different approach to calculating dependences. Instead of using a single structure, and tracking dependences in chains, we use a decentralized/distributed storage to track runtime dependences. We call this distributed storage, sets. Using static analysis, we are able to figure out which instructions are likely to conflict (i.e. have the same address). Hence, we can use the sets as runtime storage of the addresses of these instructions and use set operations to figure out if there are any conflicts (dependences). The key property that we utilized here is that typically a given instruction can only ever conflict with a subset of the rest of the instructions in the program. This implies that
we can design storage for each group of instructions that are likely to conflict with one another.

We find such group of instructions using set formation, and subsequently allocate sets using novel set allocation techniques. Briefly, we record relevant queries either through heuristics or from optimizations. The relationships between these queries are analyzed, and a greedy set allocation scheme quickly assigns storage in the form of sets. Based on this set allocation, set operations are chosen.

1.3.1 Signatures
The set operations that we use here include insertion, disambiguation and membership check. These operations can be quickly performed at runtime using software/hardware signatures. We evaluate our profiler using software signatures (simplified bloom filters) and a hardware signature cache. Signatures provide concise and imprecise storage for the sets. We show that through smart allocation schemes, the imprecision can be limited without necessarily sacrificing performance.

1.4 Feedback Directed Optimizations
Feedback Directed Optimizations (FDOs) utilize runtime information to optimize the code. These optimizations could be speculative or non-speculative. Here, we test the effects of feeding back the edge and data dependence profiles into a state-of-the-art compiler and making the runtime information available to traditional optimizations. We explore several processor configurations and try to analyze the effects of the changes made by runtime information in a traditional optimization framework. We also analyze the runtime information itself to understand the types of optimizations that might be able to utilize the data more effectively. In this vein, we propose control and data flow optimizations which could enable the framework to better harness the profile information provided.
Chapter 2

Background

This chapter provides background information which is helpful in understanding the later chapters. An basic introduction to alias analyses and their relation to data dependence profiling is provided. Subsequently, bloom filters and signatures are discussed. The set-based profiler relies on software signatures to realize set operations at runtime.

2.1 Alias analysis - data dependences

Alias analysis tries to determine if two or more memory operations can ever access the same address at runtime. In other words, alias analysis tries to determine if the address operands in each of the instructions are always equal, i.e. aliases. Using static analysis, alias analyses typically provide one of the three following answers.

- The instructions in question do not alias, referred to as no-alias
- The instructions in question may alias on certain occasions, referred to as may-alias
- The instructions in question always access the same address, referred to as must-alias

Optimizations query the alias analyses while performing code transformations, in order to guarantee that semantics are not violated. For example, a load instruction which accesses the same address as the immediate prior store could be removed if both the instructions always hold the same value, i.e. if they must-alias. In the case where the load and store never have the same value, the two instructions could be re-ordered with respect to one another, opening up a degree of scheduling freedom, which could have a domino effect on the slice of the load. However, if the alias analysis responds with a may-alias, then the optimization cannot assume any particular relation between the load and the store. Given that it is hard to statically analyze the memory behavior of irregular programs, alias analyses respond with a may-alias quite frequently. Prior works have highlighted this problem, and our studies confirm the same.
The *may-alias* response means that the alias analysis is not able to ascertain a definite relationship between the instructions in question. A data dependence profile can quantify the relationship between the two instructions, by calculating the probability that the instructions in question share a dependence. This information is valuable to optimizations which can now optimize the code, albeit with speculative information. The profiling information is speculative since it is only valid for the given inputs, and even on other correlated inputs, the behavior might be vary on occasion. Hence, any transformations using data dependence profiling are speculative. In case of SpeculativePRE [33], this speculation may not result in loss of semantics, but a loss of performance.

Figure 2.1 shows a control-flow graph with only the relevant memory operations and branches. To simplify analysis, we remove the branch instructions, while maintaining the structure of the control-flow. This simplified structure is shown in Figure 2.2. As mentioned previously, optimizations query alias analyses regarding relationships between memory operations. Here, we show the *may-alias* responses from the alias analysis. The dotted lines represent the queries for which may-alias responses were returned. Figure 2.3 shows the runtime information recorded by a DDP. It shows that instructions 1 – 3 access the same address (A), whereas instructions 4 – 6 access the address B. Therefore, we can bin these instructions based on their access patterns, as shown in Figure 2.4(a). Figure 2.4(b) shows the result of optimizing the code based on the newly available information. Optimizations can utilize this information to replace stores 2 and 3 with store 7. Load 4 can be moved into the same block as load 1, as load 4 doesn’t alias with
Figure 2.2: Responses from Alias Analysis

Figure 2.3: Runtime information gathered by a data dependence profiler
Figure 2.4: Profile Data classification and transformation
the stores 2 and 3. Since load 4 and load 5 access the same address without an intervening store, load 5 is redundant and hence removed. Therefore, in this example, DDP information was able to remove 3 memory operations.

Since the number of data dependences within even medium-sized programs are often quite large, smart strategies are required to quickly and accurately gather this runtime information. In Chapter 4, we discuss how hash-table profilers try to solve this problem, and in Chapters 5 and 6, we describe how a novel set-based profiler efficiently profiles large programs.

2.2 Bloom filters and Software Signatures

We employ software signatures to perform the insertion, disambiguation and membership operations on the sets of instructions. Here, we describe the design of the software signatures, and their roots in hardware signatures and bloom filters.

Bloom filters [4] provide concise storage of data in a bit-vector while providing high accuracy for disambiguation and membership operations on the recorded data. Unlike other compression techniques, the inserted data cannot actually be retrieved, but rather can only be queried for existence. On such queries (intersection or membership), Bloom filters are able to answer with high accuracy whether that input was ever inserted in the filter. The answer is always conservative by design—it must answer yes if the data was ever inserted. If it answers no, then the data was never inserted. However, it may answer yes due to collisions even when the data is not present. Hence, Bloom filters suffer from false positives but not false negatives.

Bloom filters consist of one or more hash functions which are linked to a single/partitioned memory for storing data. The hash function(s) set bit(s) in the memory provided. Multiple hash functions provide resilience against false positives. Previous studies [15] have shown that using partitioned memory results in better performance versus accuracy tradeoffs in bloom filters.

Signatures are simplified bloom filters. They are essentially Bloom filters employing simpler hash functions and are tied to dedicated registers. In order to adapt them to hardware, the hash functions needed to be simplified, and the number of bits used were limited by the size of registers that can be feasibly implemented. These hardware derivations of bloom filters have been used in implementing memory consistency [6], memoization [29] and load-store queue disambiguation [27]. We found that these hardware signatures provide high accuracy on large sets of addresses, and since they were intended for hardware implementation, their runtime costs were low. For these reasons, we decided to implement them in software for the set-based profiler that we propose here. As our studies show, the software signatures provide a better tradeoff between accuracy and performance when compared to the current state-of-the-art hash table implementations.

We perform three types of operations on the software signatures presented as shown in the stores 2 and 3. Since load 4 and load 5 access the same address without an intervening store, load 5 is redundant and hence removed. Therefore, in this example, DDP information was able to remove 3 memory operations.

Since the number of data dependences within even medium-sized programs are often quite large, smart strategies are required to quickly and accurately gather this runtime information. In Chapter 4, we discuss how hash-table profilers try to solve this problem, and in Chapters 5 and 6, we describe how a novel set-based profiler efficiently profiles large programs.

2.2 Bloom filters and Software Signatures

We employ software signatures to perform the insertion, disambiguation and membership operations on the sets of instructions. Here, we describe the design of the software signatures, and their roots in hardware signatures and bloom filters.

Bloom filters [4] provide concise storage of data in a bit-vector while providing high accuracy for disambiguation and membership operations on the recorded data. Unlike other compression techniques, the inserted data cannot actually be retrieved, but rather can only be queried for existence. On such queries (intersection or membership), Bloom filters are able to answer with high accuracy whether that input was ever inserted in the filter. The answer is always conservative by design—it must answer yes if the data was ever inserted. If it answers no, then the data was never inserted. However, it may answer yes due to collisions even when the data is not present. Hence, Bloom filters suffer from false positives but not false negatives.

Bloom filters consist of one or more hash functions which are linked to a single/partitioned memory for storing data. The hash function(s) set bit(s) in the memory provided. Multiple hash functions provide resilience against false positives. Previous studies [15] have shown that using partitioned memory results in better performance versus accuracy tradeoffs in bloom filters.

Signatures are simplified bloom filters. They are essentially Bloom filters employing simpler hash functions and are tied to dedicated registers. In order to adapt them to hardware, the hash functions needed to be simplified, and the number of bits used were limited by the size of registers that can be feasibly implemented. These hardware derivations of bloom filters have been used in implementing memory consistency [6], memoization [29] and load-store queue disambiguation [27]. We found that these hardware signatures provide high accuracy on large sets of addresses, and since they were intended for hardware implementation, their runtime costs were low. For these reasons, we decided to implement them in software for the set-based profiler that we propose here. As our studies show, the software signatures provide a better tradeoff between accuracy and performance when compared to the current state-of-the-art hash table implementations.

We perform three types of operations on the software signatures presented as shown in
Algorithms 1 through 3.

**Algorithm 1: Insertion**

1. \[ h = \text{Hash(Addr)}; \]
2. \[ \text{index} = h/32; \]
3. \[ \text{bitpos} = h \mod 32; \]
4. \[ S[\text{index}] = S[\text{index}] \| (1 \ll \text{bitpos}); \]
5. return \( S \)

**Algorithm 2: Disambiguation**

1. conflict=false;
2. for \( i=0; i<M; i++ \) do
3. \[ \text{conflict} = \text{conflict} \| (S_1[i] \| S_2[i]); \]
4. end
5. return conflict

**Algorithm 3: Membership Check**

1. \[ h = \text{Hash(Addr)}; \]
2. \[ \text{index} = h/32; \]
3. \[ \text{bitpos} = h \mod 32; \]
4. return \( S[\text{index}] \& (1 \ll \text{bitpos}) \)

### 2.3 Signature Operations

For a set-based profiler to be fast, its underlying set implementation must be very efficient. Signatures are one such approach. They typically use multiple hash functions to store elements in one or more bit arrays. Signatures provide three types of operations, namely insertion, disambiguation and membership, shown in Algorithm 1,2,3. An insert operation adds a single element to a given set, while the disambiguate operation checks for common elements between
two sets. The membership operation checks if a single element is part of a set.

The accuracy and performance of signatures are important for the overall success of our set-based profiler. Since signature operations are imprecise, they may result in false positives. It is well known that keeping signatures relatively underpopulated for their size is important for attaining good accuracy, as shown in the following formula:

$$ P_{\text{NoConflict}} \propto \left(1 - \frac{1}{\# \text{ of bits in } S}\right)^{kn} \quad (2.1) $$

where $n$ is the total size and $k$ is a constant determined by the hashing function.

We can understand the runtime costs by the following formula:

$$ \text{Overhead}(S) = f_{\text{alloc}}(S) \times N_{\text{alloc}} + f_{\text{insert}}(S) \times N_{\text{insert}} + f_{\text{disamb}}(S) \times N_{\text{disamb}} + f_{\text{member}}(S) \times N_{\text{member}} + N_{\text{bloat}} \quad (2.2) $$

where $f_{\text{alloc}}(S)$, $f_{\text{insert}}(S)$, $f_{\text{disamb}}$, and $f_{\text{member}}$ are the costs of allocation, insertion, disambiguation and membership operations on the signature, and $N_{\text{alloc}}$, $N_{\text{insert}}$, $N_{\text{disamb}}$ and $N_{\text{member}}$ are their dynamic counts. $N_{\text{bloat}}$ accounts for the deterioration in performance of the surrounding code due to optimization hurdles created by the signature. For signatures allocated on the stack, $f_{\text{alloc}}$ accounts for initialization costs. $f_{\text{insert}}$ and $f_{\text{member}}$ are roughly equal in magnitude, and vary depending on the size of the signature and hash functions used. $f_{\text{disamb}}$ increases as the size of the signature increases. All dynamic counts are based on actual program behavior.

**Membership vs. Disambiguation.**

Looking at the behavior and cost of signatures, some optimizations are evident. Since a single disambiguation is more expensive than a membership check, using membership operations to compare one set against another can decrease costs in some cases. For example, if one set has only a single member, it makes more sense to perform a membership operation. More generally, membership operations are more economical on a small set and disambiguation is more economical on large sets. We find that membership is usually preferred for the relationships queried by optimizations performing code motion (like Loop Invariant Code Motion (LICM)), since these queries require comparing one instruction with multiple prior instructions.
**Signature Configuration**

The size of a signature and the configuration of the hash functions also affect the cost of signature operations. Hash functions that are well randomized are good for accuracy but may hurt performance. Furthermore, using multiple hash functions to set multiple bits increases instruction count per signature operation but improves accuracy. Similarly, choosing signatures that are too large will slow down the profiler without providing additional accuracy. Signatures that are too small may be faster but result in inaccuracy due to false positives.
Chapter 3

Methodology

3.1 Host System and Benchmarks

The profilers were tested natively on an Intel Xeon two-way hyper-threaded quad-core machine with 6GB of RAM, and running RHEL 5. Eleven benchmarks from the SPEC2000 benchmark suite were used to evaluate the profilers. The fortran benchmarks in the suite were converted to C using f2c.

For the FDO framework, we simulated representative regions of the benchmarks (obtained through SESC [25] simulation marks), using PTLSim [35], a cycle-accurate x86 simulator. The trigger mechanism in PTLSim is used to select the regions identified by SESC simulation marks. The processor configurations used for the PTLSim runs are listed in Table 7.1. The four configurations listed are meant to represent various design points, with different levels of aggressiveness in terms of structures and out-of-order behavior. The IPCs for the SPEC2000 benchmark, 164.gzip, are listed in the last row, as a reference. The Default configuration is the most aggressive configuration, whereas Config 2 is the least aggressive. These configurations are used to understand the sensitivity of the feedback directed optimizations to the underlying hardware.

3.2 Compiler

The data dependence profilers are implemented as IR-level passes in LLVM 2.4 [19].

Traditional workflow for an LLVM compiler is shown in Figure 3.1. The LLVM bytecode is the intermediate representation (IR) for LLVM, and can be obtained from the source code using either the LLVM-GCC frontend or the Clang [1] frontend. Subsequently, the bytecode is optimized using the IR-level passes (like LICM, GVN and PRE). Finally, the bytecode is lowered to assembly using the backend provided by LLVM.
Table 3.1: Processor Configurations

<table>
<thead>
<tr>
<th>Specification</th>
<th>Default</th>
<th>Config 2</th>
<th>Config 3</th>
<th>Config 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Queue</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Max Issue Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Phys. Reg. File</td>
<td>256</td>
<td>64</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>ROB Size</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>In-flight Branches</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Load Queue</td>
<td>48</td>
<td>12</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>Store Queue</td>
<td>32</td>
<td>8</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>32</td>
<td>8</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Fetch Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Frontend Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Frontend Stages</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Writeback Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Commit Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Forwarding Latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Gzip IPC</td>
<td>1.1</td>
<td>0.6</td>
<td>0.93</td>
<td>0.82</td>
</tr>
</tbody>
</table>

Figure 3.1: Typical workflow in LLVM

Figure 3.2: Workflow with Data Dependence Profile
3.2.1 Data Dependence Profilers

The workflow for the data dependence profiler is described in Figure 3.2. The set-based profiler is implemented as a sequence of passes within LLVM, which operate at the IR-level. The output of these passes is instrumented bytecode. The passes provide a wide variety of instrumentation options which can be configured on the command line. The passes were invoked using the `opt` tool in LLVM.

The set-based profiler and the traditional (hash table) profiler are implemented as separate passes. We evaluate the impact of using the set allocation schemes on the hash table profiler as well. Each of the profilers uses the same bookkeeping code, to update counters to track aliases at runtime. Both the profilers output files in the same format, so a single feedback pass at the IR-level can understand the output of both the profilers.

**Ideal Profiler** This profiler calculates perfectly accurate profiles. Unlike the set-based or traditional profiler, this profiler does not lose precision. However, as a tradeoff, this profiler runs orders of magnitude slower than either of the two profilers evaluated here. This profiler was implemented as a gold standard for accuracy. It uses a large amount of memory to track runtime information, and couldn’t scale to some of the larger benchmarks, like gcc, and hence they were omitted from the results.

3.2.2 Feedback Directed Optimization

The workflow for the FDO framework is shown in Figure 3.4. The FDO component, and the associated profilers were implemented as a standalone tool compatible with LLVM 3.1. The profiler uses the same techniques as the one implemented in LLVM 2.4. The profiler in this
Figure 3.4: Workflow with DDP and Edge Profile feedback to FDO
version of LLVM uses the same techniques as the one in LLVM 2.4. In addition to the data dependence profiler, the edge profiler is incorporated in the tool, to provide runtime control-flow information. This information is used by the Dead Block Elimination pass to straighten the code, and remove unused blocks. This pass cleans up the code significantly. The workflow for the edge profile feedback is shown in Figure 3.3.

The straightened, optimized bytecode is now fed into the set-based profiler for instrumentation. Subsequently, it is lowered to binary, and profile data is collected on a train input. This data is fed back to the compiler, along with the edge profile data that is already available. The feedback pass feeds the useful parts of the data dependence profile information into a speculative alias analysis. This alias analysis makes the runtime information available to all the optimizations that are performed on the bytecode. The output of this stage is speculatively optimized bytecode which is lowered to a speculative binary. Since we do not model recovery, the speculative binary is run on the same train input, and representative regions are simulated on the PTLSim configurations shown above.
Table 3.2: Profile Coverage - Loads and Stores profiled

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>%of Loads</th>
<th>%of Stores</th>
<th>%of Fns</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>79</td>
<td>80</td>
<td>62</td>
</tr>
<tr>
<td>swim</td>
<td>78</td>
<td>78</td>
<td>100</td>
</tr>
<tr>
<td>mgrid</td>
<td>79</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>applu</td>
<td>47</td>
<td>64</td>
<td>75</td>
</tr>
<tr>
<td>vpr</td>
<td>82</td>
<td>87</td>
<td>77</td>
</tr>
<tr>
<td>mesa</td>
<td>63</td>
<td>79</td>
<td>78</td>
</tr>
<tr>
<td>art</td>
<td>83</td>
<td>81</td>
<td>80</td>
</tr>
<tr>
<td>ammp</td>
<td>78</td>
<td>86</td>
<td>88</td>
</tr>
<tr>
<td>parser</td>
<td>84</td>
<td>90</td>
<td>77</td>
</tr>
<tr>
<td>bzip2</td>
<td>97</td>
<td>86</td>
<td>100</td>
</tr>
<tr>
<td>twolf</td>
<td>80</td>
<td>93</td>
<td>89</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>77</strong></td>
<td><strong>80</strong></td>
<td><strong>84</strong></td>
</tr>
</tbody>
</table>

3.3 Speculative Code Motion

Prior works on data dependence speculation have relied on Speculative Code Motion (SCM) to deliver performance improvements. SCM is an important optimization, since it can directly impact the execution time, as well as enable other optimizations to find more opportunity. Load re-ordering is an important part of SCM since increasing the scheduling freedom of loads allows slice instructions to enjoy the benefits as well. Figure 3.5 shows two examples where determining the relationship between a load and other stores within the function can enable LICM and memoization passes to find additional opportunity. In part(a), moving ld X out of the loop decreases its dynamic instruction count, and consequently allow LICM to find other invariant instructions. In part(b), if ld X doesn’t alias the other stores within the function, the function could be declared pure, and hence, memoizable. Performing these optimizations requires runtime information showing the likelihood of the relationships.

Algorithm 4: GenQueries

<table>
<thead>
<tr>
<th>Input: P: Program being analyzed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: Relevant queries</td>
</tr>
<tr>
<td>1 foreach function ∈ P do</td>
</tr>
<tr>
<td>2     foreach load ∈ function do</td>
</tr>
<tr>
<td>3         foreach store that precedes load on some path do</td>
</tr>
<tr>
<td>4                 if load may-aliases with store then</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
We try to collect this information using a profiler, for every load in the program, at the function level. We use the analysis shown in Algorithm 4 to collect the relevant queries. Every load in the program is checked against prior stores (on any path) for a may-alias. Using the alias analysis to choose the queries reduces the profiling waste by removing useless queries. The queries cover more than 80% of the loads and stores in the benchmarks, as shown in Table 3.2. Such high coverage not only highlights the importance of this optimization, but also shows that tests the profiler’s scalability to large inputs.

3.4 Sampling

We add sampling support for all the profilers evaluated here. Sampling is provided by generating two versions of every function: one with profiling support, and one without profiling support. The sampling rate determines how frequently the instrumented function is executed. In order to provide fair comparisons across different profilers, we enforce policies that guarantee equivalent work is always done. For a sampling rate of M, we always profile the first execution of a function; then we skip the next M-1 invocations. Since some functions dominate execution on their first invocation, we cannot use down sampling to make profiling arbitrarily cheap, but we can evaluate its impact on accuracy and execution time. Two rate-identical sample runs execute the same sections of the code, in this technique.

3.5 Performance and Accuracy Metrics

Performance

Performance is measured as the slowdown compared to the native runtime of an 03-optimized, uninstrumented code.

Accuracy

We use Normalized Average Euclidean Distance (NAED) for our accuracy measurement. The metric was used in [28]. Here, we describe how NAED is calculated.

The data dependences detected during the profile run are placed in a vector. A position in the vector identifies a unique static data dependence, and the value of the vector at that position is the probability of occurrence. This probability is calculated by dividing the frequency of occurrence of the data dependence in the sample by the frequency of execution of the relevant program edge computed using edge profiling.

Given two vectors computed from different profiling runs, the Euclidean distance (ED) between two such vectors is calculated as the distance between components along corresponding dimensions. Average ED (AED) is calculated by dividing ED by sqrt(n), where n is the vector length. Since each entry in the vector has a range between 0 and 1, the range of the average ED is also between 0 and 1. Hence, the average ED in this case is referred to as Normalized AED (NAED). The equation for calculating NAED is given below, with $p_i$ and $b_i$ being two profile
The best accuracy using this measure is 0 and the worst is 1. Here are some examples to help think about our accuracy measure. If every dimension were off by 50% and there were 100 dimensions, then the accuracy is worsened by a distance of 0.5. If 10% of the dimensions were completely wrong (100% off) but the rest were perfect, this has a distance, or accuracy, of only 0.316.

Figure 3.6 shows the variation of NAED under different inputs. The x-axis shows the inputs, and the y-axis represents the NAED calculated using Equation 3.1. The inputs are represented as \( i \) by \( j \), where \( i \) is the percentage of elements which are away from the reference probability by \( j \) percent. For example, 10 by 10 implies that 10 percent of the elements in the profile vector generated, vary by 10 percent from the ideal profile generated. Hence, in a profile vector of 10 elements, which contains all zeroes (0s), one element of the current profile contains 0.1, and the rest of the elements are 0s. The NAED calculated for between the two vectors is 0.032, as seen by the first point in the figure.

The set-based profiler’s accuracy lies between the two green points marked on the plot (0.164, 0.253), whereas the traditional profiler’s accuracy lies between (0.354, 0.465). In the subsequent chapters, we explain the reasons for this difference, and how the set-based profiler is able to achieve better performance than the traditional profilers while maintaining this accuracy.
Chapter 4

Traditional Data Dependence Profilers

This chapter details the design of traditional data dependence profilers (DDPs), and evaluates their performance on the SPEC2000 benchmark suite. Subsequently, the chapter explores the drawbacks with the design and implementation of profilers. The chapter serves as a motivation for the novel set-based profiler which is described in the following chapters.

Since the traditional DDPs typically use a hash-table for storing runtime information, this class of profilers are also referred to as hash-table profilers. Let us discuss the design of traditional DDPs.

4.1 Design

4.1.1 Original Design

To recap, the goal of a data dependence profiler is to gather accurate information regarding data dependences at runtime, and feed it back to the compiler. The profiler design described here was discussed in [9].

Static analyses in compilers are unable to ascertain all the memory-carried dependences within a given program. Hence, DDPs are used to provide information regarding these dependences. Traditional DDPs track multiple instructions communicating through a given memory location, by maintaining a list (chain) of instructions that access the memory location. For example, if in a given program, instructions 1, 3 and 4 access the same memory location, in that order, then a traditional DDP would form a chain consisting of $1 \rightarrow 3 \rightarrow 4$ for that memory location.

Hence, during the profile run, at each memory operation, a single instrumentation call to...
access the storage structure is added. This instrumentation call looks up the entry in the hash
table (storage) using the accessed address as the key. Subsequently, a unique identifier associated
with the instruction (referred to as Reference ID, RefID for short) is added to the list at the
given entry in the table. At the end of the profile run, the entire storage is dumped to memory,
and fed back to the compiler.

Since each address needs a dedicated entry in the structure, in order to accurately form the
dependence chains, the size of the structure can be computed as follows:

\[
\text{Size(inbytes)} = \left( \sum_{\text{MemoryOps}} \text{DynamicInstrCount}_i \right) \times (\text{sizeof(RefID)})
\]

Given that the number of memory locations accessed by the program are large, this structure
is not scalable. This was the key reason for limiting the size of the structure. However, this
restriction introduces false dependences, as instructions referring to multiple locations shared
the same entry.

This original design was proposed by Chen et al. in [9].

4.1.2 Streamlined Design

The design of this DDP profiler is streamlined using static analysis and smart table manage-
ment. Using static analysis, the precise memory operations that need to be instrumented are
identified. As was explained in Chapter 2, alias analyses can ascertain that certain memory
operations definitely point to the same location, or that they never point to the same location.
These relationships are discarded. Further, only those relationships relevant to optimizations are
profiled. This is a novel approach called optimization-driven data dependence profiling. Since
the profiler is being given speculative code motion(SCM) queries, it is re-designed to collect
precisely this information.

The major design change here is the removal of the dependence chains. SCM queries require
the profiler to determine the probability that a load can be hoisted above prior stores. To
find this probability, the profiler only needs to track if a load and any prior store in the same
region (function) share the same location. To accommodate this, the instrumentation is modified
such that each store sets a bit in the table, using its access address. Each load monitors the
structure, at each access, to check if a prior store has accessed the memory location. A separate
structure is used for tracking the number of times each load conflicted with a prior store. The
load accesses this structure using its RefID. The size of the structure in this design can be
computed as follows.

\[
\text{Size(inbytes)} = \frac{\text{MemLocationsAccessed}}{8}
\]

The size of the structure is limited to the number of MemLocationsAccessed, and unlike the
Table 4.1: Changes in design to streamline the profiler

<table>
<thead>
<tr>
<th>Original Design</th>
<th>Streamlined Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure records dependence chains</td>
<td>Structure records prior store access</td>
</tr>
<tr>
<td>Each instruction enters ReffID in structure</td>
<td>Stores set a bit. Loads check for a bit</td>
</tr>
<tr>
<td>Dependences between consecutive instructions accessing the same address, are recorded in structure.</td>
<td>Dependences provided by static analysis are profiled. Precise dependence tracking possible.</td>
</tr>
<tr>
<td>Deducing other relationships (beyond nearest instruction) requires costly (and imprecise) static analysis in the feedback pass.</td>
<td>Feedback does not require further analysis as relevant queries are instrumented by profiling pass</td>
</tr>
</tbody>
</table>

prior design, each entry only occupied a single bit. Hence, the division by 8 to convert bits to bytes.

We find that when allocating a hash table at a function granularity (a hash table allocated on the stack, on each function invocation), the streamlined design is able to scale to larger programs when compared to the original design. However, the cost of clearing the table increases as the size of the table increases. Hence, the performance of the profiler severely degrades with increase in the size of the table.

4.1.3 Improved Original Design

In our tests, we found that the original design orders of magnitude slower, and far more inaccurate than the streamlined design. Hence, we decided to improve the original design using static analyses to determine the queries to be profiled. These improvements bring the original design on a level footing with the streamlined design.

As in the streamlined design, we use the SCM queries along with static analysis to narrow down the number, and type of queries to be profiled. The size of the table is restricted, in order to scale to larger programs. The table stores tags of previous instructions that accessed this location, as opposed to bits that are used in the streamlined design.

We evaluate the improved original design and the streamlined design in the next section. In the subsequent chapters, we evaluate the impact of set-allocation schemes on these designs.

4.2 Evaluation

4.2.1 Streamlined Design

Table Sizing

Figure 4.1 shows the performance and accuracy for a function hash table profiler using Knuth hash functions. The size of the hash table is varied from $10^4$ to $10^5$. The performance degrades
Figure 4.1: Impact of size on Streamlined Hash Tables

by about 3.5x while the accuracy improves by 0.04 NAED. The performance degradation is
caused due to the higher memory footprint of the larger hash tables, which results in higher clearing costs and lesser locality. The accuracy improves, but at a much smaller rate than the degradation in performance.

**Table Sampling**

Figure 4.2 shows the performance and accuracy plots for a hash-table profiler which uses the Knuth hash function [18]. This is a function level hash table profiler, where each function has a stack allocated hash table on each invocation. The table contains 50,000 entries. The figure shows the variation in performance and accuracy when the sample rate is increased from 1 (no sampling) to 100. The performance improves by 3x - from 6x to 2x - while the accuracy doesn’t change. There are two observations that can be made from these numbers.

- The accuracy of the profiler does not appear to degrade with increased sampling. In fact, sampling appears to have little effect on accuracy.
- In the sampling technique adopted here, the performance improvement from higher sampling rates appears to taper off around sample rate of 100.

This is similar in design to the profiler proposed by Chen et al. in [9]. The profiler has been tailored, as described in Section 4.1.2.

Figure 4.3 shows the performance and accuracy variation through sampling, for hash tables of size $10^5$. The larger size of the table leads to higher operations costs, but the pattern seen in the previous figure is reinforced here. The accuracy improves negligibly while performance improves significantly.

Figure 4.4 shows the performance and accuracy variation through sampling, for a hash table of size $2 \times 10^5$. The performance and accuracy for parser are zeroes because it runs out of memory when using a hash table of this size. The fact that sampling doesn’t improve or deteriorate accuracy leads to two important observations.

- The sampling technique shown here does in fact retain coverage, and provide a representative picture even at higher sampling rates.
- The loss of accuracy in the hash tables occurs uniformly across function calls, rather than on a small subset of the calling contexts for a given function.

Both these observations are corroborated by the set-based profiler, which is able to profile these programs accurately and quickly, leveraging the sampling technique shown here.
4.2.2 Improved Original Design

Here, we show the performance and accuracy numbers for the improved original design, varying the number of tags at each entry. Each of the tags is used to record the address of the prior store.
that accessed this location. Therefore, given 5 tags at a given entry in the table, a maximum of five unique addresses of prior instructions can be recorded. This improves the accuracy of the
profiler by providing a protection against collisions in the hash table. To ensure conservative behavior, when all the tags at a given entry have been used, we set the overflow bit. This
behavior ensures that this design is at least as accurate as the streamlined design, in every case.

Figures 4.6 and 4.5 show the results when using 10 and 20 tags per entry respectively. The bars represent the total memory used by the hash table (including tags). Therefore, the

![Graph of performance and accuracy](image-url)

(a) Performance

(b) Accuracy

Figure 4.5: Tagged Hash Tables with 20 Tags per entry
number of entries for the hash table can be calculated from the Size and Num Tags Per Entry as follows.

\[ \text{NumEntries} = \frac{\text{Size}}{\text{NumTagsPerEntry}} \]

(a) Performance

(b) Accuracy

Figure 4.6: Tagged Hash Tables with 10 Tags per entry
In both the runs, 197.parser runs out of memory at 100K run itself, hence the results show zeroes for this benchmark. 300.twolf runs of memory in the 300K configuration (300,000 entries in the hash table). These results show that the performance of the hash table profiler degrades by about 1.5x at the top end (300K), while the accuracy improves from 0.46 to 0.35. The degradation in performance can be linked to the cost of inserting and checking the tags. These operations necessarily introduce a loop over the number of tags in both the insertion and membership operation.

4.3 Drawbacks

The design of hash table profiler is simple to understand, and requires simple instrumentation in the pre-profile run phase. However, the profiler has quite a few drawbacks which make it unsuitable for profiling code with complex memory access patterns, and large memory footprints.

4.3.1 Original Design

In the original design of the profiler, the memory costs increases dramatically as the number of instructions executed increases. The operations costs increase as well, since neither the hash table, nor the extendable lists provide high locality in caches. Since a virtually unlimited size structure is not feasible, in terms of memory and operations costs, most implementations restrict the amount of memory used through a fixed size table. This affects accuracy in unpredictable ways, and due to the lack of any other safety hooks in the design, this deterioration cannot be controlled. Since dependence chains are the sole source of data dependence information in this system, inaccuracy in their calculation decreases the usefulness of this data to optimizations.

Even assuming the correct calculation of dependence chains, the design only records the order of accesses to a given memory location. Performing relevant queries like those in Speculative Code Motion is very tough or impossible in this design. For example, given a loop, checking a load against other stores within the loop requires precise answers from the profile. In this case, that is complicated by the presence of loads and instructions from other functions which may be accessing the same memory location. This situation is made worse if the dependence chains are not calculated accurately.

4.3.2 Streamlined Design

This design has been developed to leverage compiler analyses to decrease memory and runtime costs, while providing relevant information back to optimizations like SCM. However, this design also suffers from some of the same problems as the prior design. Fixed-size tables are necessary
since unlimited size tables are infeasible. This leads to inaccuracies which are unpredictable, as they are a product of the memory accesses and conflicts caused due to hash function.

4.3.3 Improved Original Design

This design improves on the original design by introducing static analyses to enable precise profiling of queries. It differs from the streamlined design, in that it uses tags to store prior addresses, rather than a single bit. This improvement allows this design to outperform the prior designs in accuracy, while suffering from performance losses due to increased operations costs.

These designs show that it is tough to design a profiler than can quickly and accurately profile a large program. The root of the problem with the design of these profilers is the degree of freedom that they provide to each instruction. It is assumed that each instruction can access any given memory location, at runtime. This assumption needs to be made since there is no static analyses to provide more precise information. Arising out of this assumption, a central structure is the only way to support data dependence collection. To make lookups quick, the address of the instruction is used to index into the structure. This leads to a host of problems that were described above.

We found that in order to solve this problem, a novel approach to data dependence profiling was required. The centralized structure for storing runtime information was not suitable for providing quick and accurate. Instead, we developed a set-based profiler that decentralizes storage, and is able to perform quick operations on this storage. In the next couple of chapters, we describe our novel set-based profiler which is able to profile large programs efficiently.
Chapter 5

Set-based Profiling using Software Signatures

5.1 Main Idea

There are several motivating factors behind the set-based approach. First, tracking related dependences as a single set may be cheaper in terms of resources and operations performed on them. Second, by explicitly grouping related dependences together, it may be possible to improve the accuracy of profiling. Sharing resources across related dependences, which are likely to share same behavior, leads to better allocation of resources to sets, hence improving accuracy.

Finally, aggressive transformations by speculative optimizations require dependence information beyond their immediate predecessors in the dependence graph. Sets allow tracking of relationships along multiple paths starting from or leading up to the candidate instruction(s).

Our set-based profiler draws inspiration from the way code optimizers work. Often, an optimizer wants to prove the absence of a dependence between one reference and a set of other references (e.g. moving a load above some stores). For convenience, let us call the instruction that is a candidate for optimization the protagonist and the other references that must be checked antagonists. If any of the antagonists inhibit optimization, then it is not possible. Hence, tracking them jointly may be sufficient since their joint answer is all that matters.

In Figure 5.1, to calculate the dependence between the load to X and the stores in the loop, we compute \{X\} \cap \{A, B, C\}. Interestingly, this comparison computes the property needed to optimize the program, which makes it easy to feed the result of profiling back to the compiler.

We can generalize this idea for profiling arbitrary sets of information. First, we profile only the relationships required by optimizations, and these relationships are identified by optimizations prior to profiling. Second, we can cast this work as operations on sets, using simple rules that map the relationships to sets. For example, protagonists and antagonists can never be in
the same set since they must be disambiguated. But, we can clump protagonists with other protagonists and antagonists with other antagonists arbitrarily.

5.2 Set-Based Profiler Design

Figure 5.2 shows the key stages of our profiler. The profiler takes as input a set of queries from an analysis or optimization. These queries are interpreted as Query Sets (QSets). Each QSet consists of two groups of instructions, namely, the LHS and the RHS, and a region over which the two groups should be compared. In this work, we always consider the region to be the parent function body.

The QSets are analyzed and used to build an Interference Graph (IG) in the Interference Graph Generation stage. Each node in the interference graph represents a set of expressions derived from a load or store in a QSet. Edges (a.k.a interference edges) between nodes indicate that a disambiguation must be performed between the two sets; hence, they cannot be allocated to the same Profile Set (PSet). Such a lowering from QSet to IG is useful because relationships are often repeated in several QSets since individual instructions (e.g. for SCM, a store) often

![Diagram](image.png)

Figure 5.1: Examples of dependence checks for speculative optimizations.
impact the potential to optimize other instructions (e.g. for SCM, multiple different loads).

The IG is analyzed and grouped into Profile Sets (PSets) during \textit{PSet Allocation}. An expression contained in a given PSet will be inserted into that set at runtime. Any interference edge in the graph indicates a membership or intersection operation that must be scheduled at runtime. Finally, the IG and PSets are input to \textit{Instrumentation} which generates the instrumentation for profiling in the LLVM bitcode.

Throughout this section, we will use the example shown in Figure 5.1 to explain the process of instrumentation using sets. Figure 5.1 shows a set of example queries from SCM, which are interpreted by the profiler in terms of QSets. In this case, there are two queries. Loads X and Y are to be compared against stores A and B, and stores B and C, respectively. These queries are interpreted as two QSets, with the LHS and RHS representing the corresponding values to the compared, and the Loc specifying where the comparison operation needs to be scheduled. Given these QSets, the profiler analyses and instruments over three stages, namely, Interference Graph Generation, Set Allocation and Instrumentation. We will describe these three stages using the given example next.

\subsection*{5.2.1 Interference Graph Generation}

Interference Graph Generation (IGG) is the stage where QSets are changed into an interference graph based representation. Interference graphs (IGs) have been used extensively as the basis for graph coloring for register allocation. We use them for a similar purpose here. They will help use map QSets into PSets. Each node in the interference graph represents a set of addresses. Edges (a.k.a interference edges) between nodes indicate that a disambiguation must be performed between the two sets; hence, they cannot be allocated to the same PSet. We have discovered two different approaches for IGG, \textit{Basic Sets} and \textit{Root Sets}.
### Algorithm 5: Generating interference graph for basic set allocation

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>while not S.empty() do</td>
</tr>
<tr>
<td>2</td>
<td>q = S.pop();</td>
</tr>
<tr>
<td>3</td>
<td>foreach le ( \in q.\text{LHS} ) do</td>
</tr>
<tr>
<td>4</td>
<td>foreach re ( \in q.\text{RHS} ) do</td>
</tr>
<tr>
<td>5</td>
<td>insert re in interf[le];</td>
</tr>
<tr>
<td>6</td>
<td>insert le in interf[re];</td>
</tr>
<tr>
<td>7</td>
<td>insert re in interf\text{degree}[interf[re].size()];</td>
</tr>
<tr>
<td>8</td>
<td>end</td>
</tr>
<tr>
<td>9</td>
<td>insert le in interf\text{degree}[interf[le].size()];</td>
</tr>
<tr>
<td>10</td>
<td>end</td>
</tr>
</tbody>
</table>

---

**Basic Sets**

This scheme is a simple set allocation scheme which directly maps the QSets relations to the IG. Given a set of queries \( S \), the algorithm creates a unique IG node for each SSA-name, global memory location, or stack location in the program that is present in a QSet. Then, interference edges are created between the nodes of the RHS and LHS of each QSet. This algorithm splits QSets into many different IG nodes. This creates an IG with a large number of nodes and at least one interference edge per node. The hope is that *Set Allocation* can merge the large number of QSets into a small number of PSets.

There are many reasons to do it this way. First, since a given memory reference is usually involved in multiple QSets, it ensures a low insertion cost since the address is only placed in one PSet globally rather than a PSet for each QSet. Second, it allows *Set Allocation* to keep the number PSets low (\( N_{\text{alloc}} \) is kept low) because it can group nodes together that have no adjoining interference edge.

The algorithm is shown in the Algorithm 5. In the algorithm, the interference degree (interf\text{degree}) is used to track the number of interferences for each instruction. It is important to note here that each interference edge counts as a degree, even if the edge occurs multiple times in various QSets.

Applying this theory to the running example from Figure 5.1, we get the graph shown in Figure 5.3(a). The interference edges here span the LHS-RHS boundary in each of the two QSets. This scheme creates the smallest number of sets required to profile the input. In our results, we found that the sets were often overcrowded, leading to a loss of accuracy. Therefore, although this scheme gives the smallest slowdowns, we found that it lacked accuracy, and hence designed our next scheme, called Root Sets, which emphasizes accuracy over performance.
Figure 5.3: Interference Graph using proposed set allocation schemes.

Root Sets

Algorithm 6: Generating interference graph for root set allocation

<table>
<thead>
<tr>
<th>Input:</th>
<th>S: Set of Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>interf_degree: Interference Graph for Queries</td>
</tr>
<tr>
<td>Result:ler:</td>
<td>inter: Interferences for each instruction</td>
</tr>
<tr>
<td>1</td>
<td>call BasicSets(S, interf_degree, inter);</td>
</tr>
<tr>
<td>2</td>
<td>while not S.empty() do</td>
</tr>
<tr>
<td>3</td>
<td>q = S.pop();</td>
</tr>
<tr>
<td>4</td>
<td>foreach re ∈ q.RHS do</td>
</tr>
<tr>
<td>5</td>
<td>remain = q.RHS - re;</td>
</tr>
<tr>
<td>6</td>
<td>if rest ∈ remain then</td>
</tr>
<tr>
<td>7</td>
<td>if interf[re] != interf[rest] then</td>
</tr>
<tr>
<td>8</td>
<td>insert re in “interf[rest]”;</td>
</tr>
<tr>
<td>9</td>
<td>insert rest in “interf[re]”;</td>
</tr>
<tr>
<td>10</td>
<td>end</td>
</tr>
<tr>
<td>11</td>
<td>end</td>
</tr>
<tr>
<td>12</td>
<td>insert re in interf_degree[interf[re].size()];</td>
</tr>
<tr>
<td>13</td>
<td>end</td>
</tr>
<tr>
<td>14</td>
<td>end</td>
</tr>
</tbody>
</table>

In this scheme, instead of trying to group instructions together, we try to separate them into multiple sets, by default, to reduce interference. We have observed that certain expressions always appear together in QSets. These common subsets are called Root Sets. Furthermore, we want to make sure that distinct Roots are not mapped to the same PSet. We accomplish this by identifying common subsets within the QSets and adding interference edges between Root
This motivation is reflected in the Algorithm 6, and shown in Figure 5.3(b). In addition to adding interference edges between both the LHS and RHS, we place edges between members of different Root sets. In the example, this means that stores A and B can’t share a set since both appear together in the QSet 1 RHS, but not in QSet 2 RHS. This guarantees that Set Allocation will segregate the RHS into distinct PSets based on Roots. In this example, we do not need to segregate the LHS since the membership operation (described later) accomplishes this. The additional edges introduced by Root Sets are marked by double stroked arrows for convenience, they are indistinguishable in our actual algorithm.

Notice for both the algorithms that differences in implementation are transparent to the next stage. This allows us to design the set allocation and instrumentation independent of the IGG stage. However, the decisions taken at this stage affect the output for each of the later stages, resulting in different tradeoffs between accuracy and performance.

5.2.2 Set Allocation

Once the interference graph is available, the instructions can be allocated to PSets.

Algorithm 7: Allocating sets using interference graph

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
</table>
| 1    | create map “sets_allocated”;
| 2    | foreach degree ∈ interf_degree do |
| 3    | create bit vector “interf_sets” for exp1;
| 4    | foreach exp2 ∈ in_degree do |
| 5    | if sets_allocated[exp2] then |
| 6    | interf_sets = interf_sets || sets_allocated[exp2]; |
| 7    | end |
| 8    | end |
| 9    | set = findFirstFreeSet(interf_sets);
| 10   | insert exp1 in sets_allocated[set]; |
| 11   | end |
| 12   | end |

The allocation algorithm is described in the Algorithm 7, and the corresponding allocations for the basic and root schemes applied on the example are described in Figure 5.4. In addition to allocating a PSet to each instruction, we also record the PSets allocated to instructions interfering with the current one. This is important in the next stage where we instrument the code with insertion, disambiguation and membership check operations.

Part(a) in the figure shows the number of sets required to profile the example using the basic
scheme, whereas part (b) shows the same when using root sets. Here, we see the difference in the number of sets that are required to profile the example using the basic interference graph, versus the root sets version. The root sets version requires double the number of sets (four) thanks to the extra interference edges. This is typically the case when moving from basic to root sets, and it gives an early indication of the tradeoff between performance and accuracy. As a thumb rule, greater the number of edges, greater the number of sets required to profile the code.

5.2.3 Instrumentation

Finally, once the PSet allocations are complete, we can instrument the code using the set operations described in Section 2.3. Figures 5.6 and 5.5 show the instrumented code using the Algorithms 8 and 9.

Disambiguation

As described in algorithm 8, we iterate over the sets allocated, and add insert operations at each of the instructions. These operations will add the value (address accessed) to the sets at runtime. Next, disambiguation operations between the sets representing LHS and RHS of each QSet are inserted at the location previously interpreted. For each QSet, we summarize the output in the global variable $g$, which accumulates the intermediate results stored in the conflict variable. We update the global variable once per region, in our case a function call, hence the update to $g$ is placed on all function exits.
**Algorithm 8: Instrument queries using disambiguation**

```plaintext
Input: interf: Set of interf vectors generated
Input: S: Set of Queries
Input: sets_allocated: Set allocation for instructions

1. foreach set ∈ sets_allocated do
   2. create PSet "set" for set;
   3. foreach expr ∈ sets_allocated[set] do
      4. insert code "s.insert(expr)" after expr;
   end
end

6. while not S.empty() do
   7. q = S.pop();
   8. create global variable "g";
   9. create local variable "conflict=0";
   10. insert "if(conflict) g=g+conflict;" at all function exits;
   11. create vector "rhs_sets";
   12. create vector "lhs_sets";
   13. foreach le ∈ q.LHS do
      14. lhs_sets.insert(sets_allocated[le]);
   end
   16. foreach re ∈ q.RHS do
      17. rhs_sets.insert(sets_allocated[re]);
   end
   19. foreach s1 ∈ lhs_sets do
      20. foreach s2 ∈ rhs_sets do
         21. insert "conflict = s1.disambiguate(s2)" at q.Loc;
      end
   24. end
end
```

**Membership**

An alternative to using disambiguation is the membership operation. This is equivalent to inserting each instruction on the LHS into its own set, and performing a disambiguation after each one. The membership is cheaper than the insertion plus disambiguation operation, hence it is preferred. As shown in algorithm 9 it requires additional analysis to find the PSets of each of the RHS instructions. Each instruction on the LHS is checked for membership with each of these PSets. The result is accumulated in exactly the same way as the disambiguation operation above.

Comparing the instrumentation outputs for the basic and root sets, we can see that more PSets translates to more set operations at runtime. We use early termination, described later to remove redundant operations at runtime.

**5.2.4 Root Sets with Membership**

Distributing the instructions across multiple sets using root sets increases the memory footprint of the profiler, however it improves the accuracy significantly. A side effect of this allocation is the localization of sets. Sets which contain small number of instructions, are only used a
Instrumentation for Basic using Disambiguation
(a)
Instrumentation for Root Sets using Disambiguation
(b)

Figure 5.5: Instrumentation using Disambiguation

Instrumentation for Basic using Membership
(a)
Instrumentation for Root Sets using Membership
(b)

Figure 5.6: Instrumentation using Membership
Algorithm 9: Instrument Queries using Membership

Input: interf: Set of interf vectors generated
Input: S: Set of Queries
Input: sets_allocated: Set allocation for instructions

1 while not S.empty() do
2    q = S.pop();
3    create global variable “g”;
4    create local variable “conflict = 0”;
5    insert “if(conflict) g += conflict;” at all function exits;
6    foreach le ∈ q.LHS do
7        lhs_sets.insert(sets_allocated[le]);
8        insert code “sets_allocated[le].insert(le)” after le;
9    end
10   foreach re ∈ q.RHS do
11      foreach set ∈ lhs_sets do
12         insert code “conflict += set.memb(re)” after re;
13      end
14   end
15 end

Table 5.1: Profiler Configurations

<table>
<thead>
<tr>
<th>Label</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccuSig</td>
<td>Signature with Root + Membership. Size: 2K Bits</td>
</tr>
<tr>
<td>BestSig</td>
<td>AccuSig + Sample Rate: 100</td>
</tr>
<tr>
<td>FastSig</td>
<td>BestSig + Knuth Hash</td>
</tr>
<tr>
<td>AccuHT</td>
<td>FHT with Address Hash. Sample Rate: 10</td>
</tr>
<tr>
<td>BestHT</td>
<td>FHT with Knuth Hash. Sample Rate: 100</td>
</tr>
<tr>
<td>FastHT</td>
<td>GHT with Knuth Hash. Sample Rate: 100</td>
</tr>
<tr>
<td>Ideal</td>
<td>Infinite Size Hash Table w/ Collision Detect.</td>
</tr>
<tr>
<td>Baseline</td>
<td>Runtime for uninstrumented, O3-opt. code</td>
</tr>
</tbody>
</table>

small portion of the time. Hence, they do not thrash the application’s memory. We found that a majority of the instructions were assigned to these small sets. Moreover, using membership operations, we trade memory space for longer running time. A combination of root sets and membership operations, therefore, produces an accurate profiler with small active memory footprint, with many runtime operations. Fortunately, the compiler is able to schedule and optimize these operations well enough to keep the runtime costs low.

For this reason, we focus our attention on the root sets plus membership configuration for the profiler in the subsequent sections. We contrast this configuration with others when we characterize the results in Section 5.3.
5.3 Evaluation

5.3.1 Experimental Setup

We performed an extensive search of the data dependence profiler design space and came up with a few interesting designs. Table 5.1 lists the configurations with the best accuracy, the best performance and a balanced trade-off between the two, for the signature and hash-table profilers. These configurations are AccuSig/AccuHT, FastSig/FastHT and BestSig/BestHT, respectively.

As an example, AccuSig, the most accurate configuration presented in this chapter, uses Root Sets and employs membership operations to transform QSets to PSets, and the underlying signatures are 2K bits each. BestSig is AccuSig sampled at a rate of 100. FastSig combines the BestSig with a Knuth Hash, instead of the typical set of hash functions in the signatures. The table also shows the ideal and baseline profilers, which are used as references for accuracy and performance, respectively.

5.3.2 Performance and Accuracy

Figure 5.7 shows the performance and accuracy of each of the six configurations mentioned above, across a set of eleven SPEC2000 benchmarks. Accuracy, measured in terms of NAED from the Ideal profiler, is shown in Figure 5.7(b). Across the benchmarks tested, we see that the signature profilers consistently outperform the HT profilers, in many cases significantly. Considering that NAED is a non-linear scale biased towards the upper half of the range [0,1], this difference is even more impressive. On average, BestSig has an NAED of 0.15 versus AccuHT.
with 0.4. We see that FastSig is slightly less accurate than AccuHT, and we attribute this to the single Knuth hash function used for this configuration. This indicates that the use of multiple hash functions in signatures is at least partially responsible for accuracy. The profilers are completely accurate for swim and applu.

Figure 5.7(a) shows the performance (slowdown versus Baseline) for each of the configurations. The difference between FastHT and BestSig is less than 2x. BestHT and BestSig have roughly equal performance, but BestSig has much better accuracy. Interestingly, FastSig is the fastest profiler tested here. These numbers clearly show that signatures provide a better performance versus accuracy trade-off at multiple performance points. Despite our best efforts, we couldn’t get the hash table profilers to match the accuracy of the signature schemes. The reasons for this loss of accuracy have been described in Chapter 4.

**OracleSig** The final bar in each of the graphs in Figure 5.7 refers to the OracleSig configuration. In this configuration, we use the best possible set allocation scheme for each benchmark. This approach helps ammp, art and applu show impressive performance gains (up to 40x) with a small loss in accuracy (<0.05). For these three apps, we were able to trade-off the accuracy of Root Sets in favor of performance gains. We used the basic PSet allocation scheme along with membership operations to decrease overhead for an overall small drop in accuracy (<0.05).

### 5.3.3 Quality of Accuracy

Signatures are designed to provide imprecise storage which can be queried quickly. By design, the imprecision is due to false positives, and not false negatives. Hence, it is important to find out how well our signature based profiler can capture runtime no-aliases (i.e. instructions which never alias at runtime). Given that signatures suffer from false positives, this metric shows how well the addresses are distributed across signatures at runtime, although we use simple static allocation schemes. Over-used signatures lead to higher false positives.

Table 5.2 compares AccuHT (the most accurate hash table profiler) and BestSig (the most balanced signature profiler). These profilers perform equally well, hence the only difference is the accuracy. In the table, the first column for each profiler shows the percentage of No-Aliases (NAs) that were correctly captured. BestSig is able to capture almost all correctly (!) while AccuHT only manages to capture 57%. Feeding this information back to the Loop Invariant Code Motion (LICM) pass highlights the impact of accuracy on optimizations. The loads and instructions reordered by LICM, using the BestSig and AccuHT profiles, are compared with the corresponding numbers from the Ideal profiler and expressed as a percentage. The instructions reordered includes secondary effects, due to load motion. BestSig matches the Ideal profiler in almost every case, whereas the AccuHT lags significantly in almost all cases. This behavior clearly shows that the difference in accuracy (0.4 vs 0.16 NAED) has a considerable impact on
Table 5.2: Accuracy and its impact

<table>
<thead>
<tr>
<th>Apps</th>
<th>BestSig</th>
<th>AccuHT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NAs</td>
<td>Loads</td>
</tr>
<tr>
<td>gzip</td>
<td>98</td>
<td>100</td>
</tr>
<tr>
<td>swim</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mgrid</td>
<td>98</td>
<td>100</td>
</tr>
<tr>
<td>applu</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>vpr</td>
<td>97</td>
<td>100</td>
</tr>
<tr>
<td>mesa</td>
<td>97</td>
<td>100</td>
</tr>
<tr>
<td>art</td>
<td>91</td>
<td>100</td>
</tr>
<tr>
<td>ammp</td>
<td>78</td>
<td>100</td>
</tr>
<tr>
<td>parser</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td>bzip2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>twolf</td>
<td>82</td>
<td>94</td>
</tr>
<tr>
<td>Average</td>
<td>94</td>
<td>99</td>
</tr>
</tbody>
</table>

the effectiveness of the speculative optimizations performed.

5.3.4 Set Operations Counts

There are two components to consider when analyzing the signature profiler, namely, the memory costs and operations costs. Memory costs refer to the impact that the memory footprint of the profiler, on the application and on the profiler itself. Besides memory footprint, this cost also includes the memory access patterns. The memory access patterns are defined by the operations to be performed, and their placement in the code. This part is referred to as the operations costs. Here, we discuss the set operation counts for various configurations. This counts are irrespective of the underlying signature size. As we shall see in subsequent sections, the underlying signature has a considerable impact on the performance and accuracy.

Figures 5.8 and 5.9 show the insertion and membership counts, their contributions to the operations counts. In figures 5.8 and 5.9(b), the y-axis scales logarithmically. Since our scheme inserts an instruction into a single signature, the insertions costs are only sensitive to type of checking operation performed, i.e. disambiguation or membership. In disambiguation, each instruction is assigned to a signature, whereas in membership, some instructions simple check for their value in other signatures. Therefore, the number of signatures and insertions decreases when configuration decreases when configuration changes from disambiguation (S) to membership (S,M).

Membership and disambiguation counts are tricky, as they are impacted by the type of set allocation scheme and the queries provided.

**Queries** The SCM queries consist of a single load on the LHS, which needs to be checked
Figure 5.8: Set Characterization Part 1

(a) Insertion Counts

(b) Membership Counts
Figure 5.9: Set Characterization Part 2
against multiple stores on the RHS. Disambiguation groups the instructions into sets, and performs disambiguation operations. Each disambiguation operation could answer (albeit not very accurately) multiple queries simultaneously. Membership groups non-conflicting stores (if using Root Sets) into multiple sets, and checks the LHS against the sets created. No sets are allocated to the LHS instructions.

An interesting outcome of this allocation is that, as there are many more loads than stores, there are many more membership operations, as compared to insertions. And unlike disambiguation operation, multiple membership checks may be required to resolve each query. This explains the larger membership count versus the insertion count for the membership configurations – S,M, S,RM, S,RMS – in Figure 5.8.

**Basic Set Allocation** The basic set allocation (S) groups as many instructions as possible into the same set. Coupled with disambiguation, this results in greater number of insertions (as seen in Figure 5.8(a)), but smaller disambiguation counts (Figure 5.8(b)). Membership operations (S,M) exhibit the opposite trend. Insertions costs decrease as there are a smaller number of signatures, while checking costs increase since these small number of signatures are checked repeatedly at multiple locations, to gather the response for each query.

**Root Sets** This scheme tries to segregate signatures into a greater number of sets. The goal of this technique is to improve accuracy by grouping similar but non-conflicting instructions together. This significantly increases the number of signatures used. Since we only perform one insertion per instruction in the disambiguation config, the insertion counts stay the same. The disambiguation counts themselves increases since instructions are spread across a greater number of sets. This explains the increase in disambiguation counts from S to S,R.

The insertion counts for the membership config do not change, as each instruction on the RHS is still inserted into a single signature. The number of such signatures has increased however, therefore, a greater number of membership operations need to be performed in order to gather the responses for queries. Hence, the membership counts increase, sharply, from S,M to S,RM. This shows the extent to which the instructions on the RHS have been split into different signatures, compared with the basic set allocation scheme.

The role of memberships in the operations costs is clear from Figure 5.9(a). In every configuration using membership, the checking operation counts are higher than the insertion counts. The combination of membership and root sets increases the operations counts significantly. Finally, Figure 5.9(b) shows the increase in signature counts (on a logarithmic scale) across different configurations. The difference in PSets from S,M to S,RM is close to 10x (65 vs 562).

Overall, membership operations provide greater accuracy at higher operations costs. Surprisingly, Root Sets also have the same effect. Combining the two schemes sharply increases the accuracy, but with a large deterioration in accuracy. Sampling and smaller signatures are two ways of decreasing operations costs. They are evaluated and explained in the subsequent
5.3.5 Signature Sizing

Figure 5.7 shows that signatures can be configured for three different performance and accuracy tradeoffs (18x at 0.15 NAED, 8x at 0.16 NAED, and 2.97 at 0.5NAED). The only difference between AccuSig and BestSig is the sampling rate. A dimension that wasn’t explored was the size of the signature itself.

Figure 5.10 shows the performance and accuracy variation when signature sizes are varied from 256-bit to 2K. All the configurations were tested at sample rate of 1 (no sampling). Starting with the accuracy plot in Figure 5.10(b), we see that accuracy improves from 0.22 at 256-bit to 0.15 at 2K. Given that the set allocation scheme is providing all these configuration with identical instrumentations, except for the actual underlying signatures, this leads us to believe that accuracy improves as the set size increases, up to a certain point. We this leveling off around the 2K configuration. As signatures get larger (in this case, at the rate of 2x except for the 768 config), we see that the signatures grow more accurate. The reason for this improvement is the more space in each vector in each signature. As we mentioned previously, signatures are basically simple bloom filters, and they rely on multiple vectors, each assigned to one hash function, to set bits. As we shall see in Section 5.3.7, this multiple vector storage mechanism is crucial for accuracy.

Our hash functions are designed such that, as each vector increases in size, greater number of bits from the addresses can be hashed simultaneously (together) into that vector. The greater the number of bits that can be hashed at once, the lesser the change of false positives within those bits, if they are spread across different vectors. With enough number of bits (at 1K and above), the vectors share the same address bits, providing further protection against false positives.

We can also see here that benchmarks vary in their preferences for larger signatures. For example, 172.mgrid achieves its best accuracy with the 512-bit and 2K signatures. This happens because the 512-bit signature’s design is well suited to this benchmark’s access patterns. This also shows that its not always necessary that larger signatures perform better than smaller signatures. Another interesting case is 188.ammp. It has the highest slowdown among all the benchmarks shown. And its slowdown is not affected by the size of the signatures. It appears that ammp is a compute intensive benchmark, and since the operations costs across the configurations are roughly the same, the benchmark performs similarly on all the configurations. 256.bzip2 is another benchmark that exhibits similar behavior.

The interesting insight to gain from this figure is that, on average, the slowdowns versus baseline in signature configurations is due to the operations costs, rather than their memory.
Figure 5.10: Performance and accuracy comparison of signature designs
footprint. From Figure 5.9(b), we can gather that the total number of PSets (Signatures) that would be used in these benchmarks, on average, is around 550. At 256 bits per signature, this works out to about 16KB of memory, which is smaller than the L1 cache in the Xeon test machine. At 2K bits per signature, this swells to 128KB in memory (8x larger). This is smaller than the last level cache on the Xeon, and only a fraction of the total PSets is active at any given time. This is because signatures are local to functions. The number of signatures at any given time can be calculated using the following formula.

$$ActivePSets = \sum_{callstack} PSetCount_{fn(i)}$$

Hence, the actual memory footprint of the signature configuration is small, and as we shall see, they have high levels of locality. Next, we explore the role that sampling plays in the variation of performance and accuracy.

### 5.3.6 Effect of Sampling

The *BestSig* configuration uses Root Sets with Membership operations on 2K signatures. This setup profiles the program at a sample rate of 100. In other words, starting with the second call to the function, we look at every 100th call. The first call is always profiled. AccuSig has same configuration as BestSig, but has a much higher slowdown at 18x (versus 8x), and only profiles slightly more accurately (0.15 vs 0.16NAED). Here, we investigate the impact of sampling on signatures. Specifically, we show how the performance and accuracy vary with change in sample rates.

Figure 5.11 shows the variation of performance and accuracy for S_RM_2K configuration, with sampling rates varying from 100 to 1. The performance variation is subdued between sampling rates of 100 and 5, but shoots up between 5 and 1. The reason for this increase can be explained by the interaction between benchmarks and the compiler. Compilers inline functions which are called multiple times. Hence, most of the functions which are called frequently, are already inlined. The penalty for those functions is already being paid at the sample rate of 100, 50, 25, 10, 5 and 1. Since we profile the very first call to a function, we get high levels of coverage, which aids accuracy, but decreases the scope for improving performance beyond a certain point. So, a combination of the sampling technique (profiling the first call), and the compiler’s actions (in inlining the hot functions) ensures that our profiler covers the hot code at all sampling rates. The variation is significant between 5 and 1, since the functions which are called just a few times, are not inlined. Even though the function may be hot, typically compilers do not inline such functions as the added register pressure is not compensated by the function call code savings. This effect is particularly pronounced in 172.mgrid, 175.vpr, 197.parser and
Figure 5.11: Performance and accuracy variation for S\_RM\_2K using different sample rates
300.twolf. In fact, these benchmarks contribute most to the degradation in performance (from 11x at SR 5 to 18x at SR 1).

The accuracy changes negligibly from SR 100 to SR 1. There are two factors at play here. First, the sampling technique is tuned to maintain higher coverage (by profiling the first call). Second, since functions are inlined, our profiler is already profiling most of the code – at all sampling rates. 172.mgrid shows a large degradation in accuracy from SR 5 to SR 1. We attribute this to a single function in the benchmark which is called infrequently. The profiler doesn’t profile this function accurately. Lower sampling rates hide this inaccuracy, as the function is only executed once at all sampling rates except SR 1. At SR 1, all the calls to the function are executed, and there is corresponding degradation in accuracy. The opposite of this holds true in the 177.mesa and 175.vpr. Our profiler doesn’t profile the first call to a function accurately. Subsequent calls are profiled correctly, and accuracy is recovered.

In summary, although there are interesting variations as sampling rates vary, a combination of the sampling technique used and the compiler’s aggressive inlining ensure that code coverage remains high, and variation in performance and accuracy stays small.

5.3.7 Single Array Signatures

Single array signatures represent a very interesting design point for the profiler. They leverage all the set allocation schemes that promise to deliver a good performance and accuracy tradeoff on signatures, but instead of using signatures, they act more like hash-tables. Single array signatures in effect show how a small hash table would function in the set profiler. Although both hash-tables and signatures use hash functions, each of them is designed with different goals in mind. Signatures use multiple hash functions to guard against false positives. Hash-tables use a hash-function to spread the input values into different bits in the same storage.

Figure 5.12 shows the performance and accuracy plots comparing single array signatures, of varying sizes, with BestSig and BestHT. Two sizes of the single array hash table, 256-bit and 8K, are tested, and two configurations are shown for each size, namely, SR 1 and SR 100. The \( K \) in single array signature configurations refers to the Knuth [18] hash function that was used. Each of the configurations was tested to ensure that the complete signature was being utilized by the hash function.

Starting with the accuracy plot in Figure 5.12(b), we see that the accuracy for the 8K single array signature is close to that of the BestHT. On average, the NAED for the S_RM.8K configs and BestHT varies by less than 0.03. This shows that although the set allocation scheme distributed the storage into multiple sets, the sets themselves were not profiling accurately. In effect, the smart set allocation was completely undone by an inadequate storage solution. The slight improvement in performance versus BestHT can be attributed to two factors, namely,
Figure 5.12: Performance and accuracy comparison of single array signature designs
lower memory footprint and greater locality, due to the scheduling of set operations. Hash suffer from very low locality as the complete table does not fit in the cache, and even neighbouring instructions could end up generating misses. We believe this plays a significant role in the speedups achieved by signatures. However, as we shall see in the next chapter, this scheme can be improved considerably by smartly allocating, managing and realizing sets.

The S_RM_256 is the fastest signature configuration in this chapter, however, its loss of accuracy is greater than that of the hash tables, even. On a non-linear scale, in linear terms, the S_RM_256 is more than three times less accurate than the BestSig. It has a slowdown of just 2.97x compared to the baseline, which is close to three times as fast as BestSig, and more than 1.5x faster than the 8K configuration. This configuration has few memory issues, since its memory footprint is very small. From Figure 5.9(b), we see that the number of PSets (which are mapped one-to-one to signatures) for the S_RM configuration is around 550 (its a logarithmic scale on the graph). Given that each signature is 256-bits, all the signatures in the program would occupy a total of 16KB, which is smaller than L1 cache size on the Xeon machine that was used. Moreover, the PSets are local to each function, hence at any given point, only the signatures local to the current call stack are active. Hence, the only overhead in this configuration is the operational overhead. This operational overhead is decreased thanks to aggressive inlining of set operations, and a single hash insertion/membership check, versus multiple hashes for a normal signature insertion/membership check.

From a pure performance perspective, the 8K configurations have high locality with set operations while the hash-tables, on the other hand, have low locality but simple operations. Unlike hash-tables, on an average, more than one set operation is added at each profiled memory operation. Therefore, the 8K configurations have higher operations costs, whereas the hash-tables have higher locality costs. Therefore, benchmarks that are more sensitive to cache behavior than processing cycles, favor the 8K configurations. Benchmarks like 175.vpr, 177.mesa, 197.parser and 300.twolf exhibit this property. Benchmarks that are sensitive to processing cycles over cache behavior, would prefer the hash tables. Benchmarks like 179.art, 188.ammp and 256.bzip2 are examples of this behavior.

In the next chapter, we outline techniques that speedup the normal signature configuration using smart optimizations, and in fact, make it faster than the S_RM_256_SR_100.

5.3.8 Hardware Signatures

Of late, hardware signatures have been proposed in a variety of settings. In Transactional Memory [5, 34] and Thread-Level Speculation [5], they have been proposed as an efficient way to detect misspeculation and recover. Also, they have been used for checking the accuracy of speculative compiler optimizations [3]. Hardware signatures are attractive because they pro-
Figure 5.13: Impact on SpecLICM - Percent of Insts and Loads Moved
vide single cycle insertion, membership check, and disambiguation with very efficient hardware structures.

Since hardware signatures would likely be a limited resource, we would need to use them judiciously. Therefore, we could use efficient and small signatures in software for small QSets that can tolerate them, and hardware signatures can be allocated for QSets that need higher accuracy. We evaluated the performance of a 32 entry 2-Kbit signature cache, which works like a regular data cache, except that each entry is a signature. We use a random replacement policy to evict entries. Figure 5.14 shows the hit percentages for insertion, membership and the total percentage across all operations. The high hit rates show that a small hardware signature cache would perform very well, and that the locality for signatures is very high. This locality is another key to the overall performance of software signatures. Furthermore, if signatures were supported in hardware, they could play a cooperative role in detecting misspeculation during speculative optimization.

5.4 Summary

The set-based profiler uses smart allocation of decentralized storage to profile data dependence quickly and accurately. A couple of set allocation schemes were introduced, which can be mapped to a variety of underlying configurations, with varying signature sizes and operations. We found that although the design space appears large, a few designs typically deliver good performance
and accuracy tradeoffs. The set-based profiler was thoroughly evaluated to understand how performance and accuracy vary across benchmarks and configurations. The explanations show that it is possible to understand the reasons for variations in performance and accuracy using statistics such as set operation counts, types of signatures used and operations performed on these signatures.

The results in this chapter show that the smallest slowdown that can be achieved, while maintaining good accuracy, from our design space exploration, is 8x (BestSig). To put this in perspective, this slowdown is equivalent to growing the program by 8x, or in other words, adding 7 instructions for every instruction in the program. Given that the queries cover around 80% of the program, this average number of instrumentation overhead, in terms of instructions, is close to 8. This overhead is large, especially compared to that of control-flow profilers, like an edge profile.

To solve this problem, we analyze the costs in this set-based profiler, and try to understand how this profiler can be improved. Through the course of the study, we find that there are significant improvements possible to this profiler, which can bring down the overhead significantly with negligible changes in accuracy. The next chapter discusses the optimizations on the set-based profiler proposed here. The insights from the evaluation studies here were crucial in finding optimization opportunities within the set profiler.
Chapter 6

Optimizations to Set-based Profiling

6.1 Motivation

While the costs of profiling were briefly introduced in Section 5.2, we now go into more detail so that our approaches for improving performance and accuracy can be easily described in terms of their costs.

The set profiler uses smaller chunks of memory (called sets), and performs operations on these sets. Overall, the memory footprint of this profiler is lesser than traditional profilers, but its operations costs (in terms of instructions) are significantly higher. We’ll explain these costs in greater detail using the equation below.

\[
\text{Cost per Region (Set)} = \sum_{P\text{Sets}} (\text{ClearingCost}^p \times \text{RegCount}) + \sum_{\text{insts}} (\text{InsertCost}^i \times \text{InsertCnt}^i) + \sum_{\text{FinalIG}} (\text{ConflictCost}_{\text{edge}} \times \text{ConflictCnt}_{\text{edge}}) + \sum_{\text{QSets}} (\text{CounterUpdateCost}^q \times \text{RegCount})
\]

This equation gives the operational cost for a given region. In sub-equation 6.2, each PSet that is created for this region needs to be cleared at the start of the region. In sub-equation 6.3, we calculate the total cost of insertion into a PSet. For each instruction, \( i \), that computes an
Table 6.1: Characterization of Baseline Profiler.

<table>
<thead>
<tr>
<th>Bench</th>
<th>PSets Count</th>
<th>Total Clears ($\times 10^6$)</th>
<th>InsertCnt ($\times 10^6$)</th>
<th>ConflictCnt ($\times 10^6$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>86</td>
<td>987</td>
<td>2113</td>
<td>15804</td>
</tr>
<tr>
<td>swim</td>
<td>31</td>
<td>0</td>
<td>6845</td>
<td>49833</td>
</tr>
<tr>
<td>mgrid</td>
<td>23</td>
<td>2</td>
<td>5373</td>
<td>149589</td>
</tr>
<tr>
<td>applu</td>
<td>129</td>
<td>1</td>
<td>12409</td>
<td>102753</td>
</tr>
<tr>
<td>vpr</td>
<td>382</td>
<td>24543</td>
<td>6968</td>
<td>268629</td>
</tr>
<tr>
<td>mesa</td>
<td>3022</td>
<td>-</td>
<td>31595</td>
<td>58993</td>
</tr>
<tr>
<td>art</td>
<td>43</td>
<td>0</td>
<td>2593</td>
<td>129828</td>
</tr>
<tr>
<td>ammp</td>
<td>581</td>
<td>37</td>
<td>3316</td>
<td>305810</td>
</tr>
<tr>
<td>parser</td>
<td>728</td>
<td>10582</td>
<td>4600</td>
<td>77370</td>
</tr>
<tr>
<td>bzip2</td>
<td>192</td>
<td>412</td>
<td>23489</td>
<td>332136</td>
</tr>
<tr>
<td>twolf</td>
<td>975</td>
<td>9017</td>
<td>7099</td>
<td>103572</td>
</tr>
</tbody>
</table>

address that must be added into the signature, an insert operation is scheduled in the code; that insert will execute every time the instruction executes Inst$^i_{dyn}$.

Sub-equation 6.4 computes the conflict check cost. For each edge added to the IG, there must be an operation to check for a conflict between the two PSets. ConflictCost is the average cost in cycles to perform a conflict check for a particular edge. In our base design, all PSets are mapped to 2K-bit signatures, and conflict checks are usually performed using a membership operation, as in [30]. ConflictCnt is the number of times the operation will occur.

Finally, sub-equation 6.5 shows the cost of updating the global variable associated with each Query. The counter for all queries must be updated each time the region executes, RegCount. This cost is exactly the same in both Hash-table-based and Set-based profilers.

Table 6.1 shows several key cost parameters for the set-based profiler. The largest overhead currently comes from Conflict detection, which exceeds the counts of both Insertions and Clears. Hence, in terms of priority, we first focus on the number of conflict detections and insertions; then, we reduce the cost of operations on sets through signature customization.

### 6.2 Early Termination

#### 6.2.1 Set Profiler

Decreasing the number of operations needed for conflict detection can have a significant impact on performance given their large fraction of total cost. One way to achieve this with no impact on accuracy is through early termination. Since we only need to record the occurrence of a dependence once per region execution, we can stop performing conflict checks as soon as we
know the conflict is detected. Once a conflict is recorded for a given query, all other checks for that same query can be suspended.

This is easy to achieve by guarding all such conflict detection operations with a unique variable which toggles from true to false once any conflict is detected. Once toggled to false, the other operations guarded by the variable, are short circuited. This allows us to terminate the dynamic checks for this region, early. Hence, we call it Early Termination.

Applying this technique changes the cost equation by introducing a termination factor into the conflict detection cost: \( \left( \sum_{i \in \text{FinalIG}} \text{ConflictCost}_{i} \times \frac{\text{ConflictCnt}_{i}}{\text{Memoize}_{i}} \right) \). The equation has been multiplied by \( \frac{1}{\text{Memoize}_{i}} \), where Memoize denotes the fraction of checks for a given edge that have been memoized. Note that this factor does not impact the insertion, clearing or counter update costs. Given that the conflict checks are more frequent than either of these other components, we see a significant improvement in performance without sacrificing accuracy. The results are described below.

### 6.2.2 Results

The key for the configurations evaluated is shown in Table 6.2. The Baseline designs are the same as the ones used in Chapter 5. ET has Early Termination added to the Baseline; ET-S10 adds sampling at a rate of 1 in 10; ET-S100 adds region sampling at a rate of 1 in 100. Figures 6.1(a) and 6.1(b) show the performance, as slowdown, and accuracy, respectively, for the set-based profiler using early termination.

Figure 6.1(a) shows the large impact compared to Baseline. ET achieves 42% more performance over Baseline, on average. The impact on accuracy is small but non-negligible in Fig. 6.1(b). Even though conflict detection occurs over the same relationships, they may occur on different addresses. Given the probabilistic nature of signatures, this can randomly affect accuracy. In this case, accuracy was slightly worsened. However, on the whole, accuracy is very good. In fact, if you compare the impact one benchmark at a time, each benchmark, except for mesa and mgrid, exhibits the same trend as the average. Hence, across the board, early

<table>
<thead>
<tr>
<th>Key (Abbrev.)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (Base)</td>
<td>BestSig system from [30]</td>
</tr>
<tr>
<td>Baseline-S10 (B10)</td>
<td>Baseline, Sample Rate=10</td>
</tr>
<tr>
<td>Baseline-S100 (B100)</td>
<td>Baseline, Sample Rate=100</td>
</tr>
<tr>
<td>ET</td>
<td>Baseline + Early Termination (ET)</td>
</tr>
<tr>
<td>ET-S10 (ET10)</td>
<td>ET, Sample Rate=10</td>
</tr>
<tr>
<td>ET-S100 (ET100)</td>
<td>ET, Sample Rate=100</td>
</tr>
</tbody>
</table>
Figure 6.1: Early Termination
termination is highly effective for performance and accuracy.

Table 6.3 shows the impact on conflict checks. The first column shows the benchmark; the second column shows the number of conflict checks in Baseline; the third and fourth columns show the percentage of check operations that are executed as compared to Baseline-S10 and Baseline-S100. Finally, the 5th through 7th columns show the percentage of checks compared to ET, ET-S10, and ET-S100.

This table allows comparison both with and without early termination, as well as its relative impact on sampling. For most benchmarks, early termination significantly reduces the number of operations. Only mgrid, mesa, and twolf fail to see a significant (more than 10%) reduction. However, when coupled with sampling, early termination has an even bigger effect, with only mesa obtaining little benefit. However, mesa is already relatively cheap so this is not a problem. twolf, parser, and vpr perform roughly the same number of conflict detections with BestSig at a sample rate of 100 as they do with early termination at a sample rate of 100. Even though they are not helped at this sampling rate, their performance is not hurt from the extra code needed to support early termination.

### 6.2.3 Summary

Early Termination has a large impact on execution time, and as such, it should be adopted by data dependence profilers. Set-based profiling is advantageous over other forms of profilers, in this regard, since it allows many checks to be discarded as soon as a conflict is detected.

While the best performance is provided by ET-S100, the best accuracy is provided by ET. This requires us to make a trade-off between performance and accuracy, and we choose accuracy. Hence, we adopt ET as the new best design and the one to beat in later sections. However, we
see that sampling provides a big advantage if only we can conquer its inaccuracy.

After eliminating many unnecessary conflict checks, its natural to consider eliminating insertions. Insertions are a large cost and seemingly unnecessary once their corresponding membership operations are frequently memoized. We experimented with a technique that disabled insertions once all memberships that relied on that insertion were memoized. However, we found it to degrade performance across the board. The extra cost to predicate the insertion outweighed the benefit from disabling insertion.

This result is counter-intuitive. Since we are able to eliminate so many conflict checks, it would seem natural that most insertions could also be disabled. However, because of the way addresses are added to PSets, a PSet often participates in many different queries. It is unlikely that all queries associated with a PSet will be memoized at the same time. Hence, it is rare that insertions can be disabled, so the extra instrumentation cost serves merely as overhead.

Since insertions are not easily reduced, we, instead, turn our attention to the cost of set-based operations.

### 6.3 Set Customization Using Edge Profiling

Early termination reduces overall cost by reducing the number of conflict checks, but it does not change the cost of each set operation. To change per operation cost, we can tailor signatures to the characteristics of specific sets. We now focus on decreasing $Set\text{ClearingCost}$, $Insertion\text{Cost}$ and $Conflict\text{Cost}$ whenever possible, by customizing the size of each individual set.

#### 6.3.1 Estimating Population Using Edge Profiling

Different types of signatures, or maybe other types of storage which provide the functionality of sets, can provide a better trade-off on a set-by-set basis. To perform set-level customization, we need runtime information regarding the population size of each set. Using an edge profile, we are able to estimate this information, on average, using the following equation:

$$Population_{set} = \frac{\sum_{i}^{\text{insts in set}} InsertCnt_{i}}{\text{RegionCnt}}$$

(6.6)

Table 6.4 shows the percentage of sets that occur for different population counts. We show small population counts because, if they are frequent, they may be suitable for efficient representations using something other than signatures. Clearly, there is a significant number of sets with 5 or fewer elements. This suggests that a single set size of 2 Kbits is not the optimal solution.
### Table 6.4: Population Distribution Percentage

<table>
<thead>
<tr>
<th>Bench</th>
<th>Percentage of Sets with Given Pop. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>gzip</td>
<td>34</td>
</tr>
<tr>
<td>swim</td>
<td>10</td>
</tr>
<tr>
<td>mgrid</td>
<td>13</td>
</tr>
<tr>
<td>applu</td>
<td>7</td>
</tr>
<tr>
<td>vpr</td>
<td>31</td>
</tr>
<tr>
<td>mesa</td>
<td>3</td>
</tr>
<tr>
<td>art</td>
<td>40</td>
</tr>
<tr>
<td>ammp</td>
<td>13</td>
</tr>
<tr>
<td>parser</td>
<td>50</td>
</tr>
<tr>
<td>bzip2</td>
<td>27</td>
</tr>
<tr>
<td>twolf</td>
<td>33</td>
</tr>
<tr>
<td><strong>Avg</strong></td>
<td>24</td>
</tr>
</tbody>
</table>

### Table 6.5: Set Customization Types

<table>
<thead>
<tr>
<th>Population</th>
<th>Set Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32-bit Single Integer Set</td>
</tr>
<tr>
<td>2</td>
<td>64-bit Double Integer Set</td>
</tr>
<tr>
<td>3</td>
<td>96-bit Fully-Associative Array</td>
</tr>
<tr>
<td>4</td>
<td>128-bit Fully-Associative Array</td>
</tr>
<tr>
<td>5</td>
<td>160-bit Fully-Associative Array</td>
</tr>
<tr>
<td>&gt; 5</td>
<td>2K-bit Signature</td>
</tr>
</tbody>
</table>

### 6.3.2 Set Customization

Given the variance in set size and the large number of sets with population of 5 or fewer, we designed 5 new set implementations based on exact sets. Rather than use a signature, we store the addresses exactly in an array of length 5 or fewer.

Set clearing is much cheaper since we will clear a smaller amount of memory. For insertion, we insert the address into the next free element. For conflict checks, we perform a search of the full array, which amounts to several in-lined comparisons. If the array ever overflows, a conflict check is no longer exact. To retain the conservative nature of our checks, we treat an overflow as a conflict and force all future checks on the set to return a conflict. The new set implementations and allocation scheme is shown in Table 6.5.

This approach will decrease the cost of operations on small sets. However, since the population calculation is based on an average from an edge profile, it is likely that some uses of the set will overflow. If overflow is frequent, this will lead to a reduction in accuracy. If overflow is infrequent, accuracy will not be affected and performance should improve.
Table 6.6: System Configurations for Set Customization.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>ET with Set Customization</td>
</tr>
<tr>
<td>Custom-S10</td>
<td>Custom, Sample Rate=10</td>
</tr>
<tr>
<td>Custom-S100</td>
<td>Custom, Sample Rate=100</td>
</tr>
</tbody>
</table>

6.3.3 Results

The key for the configurations evaluated is shown in Table 6.6.

Figure 6.2(a) and 6.2(b) show the performance and accuracy results from set customization. Not surprisingly, this technique further improves performance. In fact, the average performance at a sampling rate of 100 has fallen below a $2 \times$ slowdown. And, compared to the results of the last section, performance is improved by 60%.

However, accuracy is worsened from 0.19 NAED to 0.33 NAED, on average. The degradation in accuracy occurs because overflow happens frequently in some of the smaller sets. Clearly, for our current implementation, set customization matters for performance but degrades accuracy. If the loss of accuracy can be mitigated, this technique will be valuable for decreasing set-operation overhead.

6.4 Improving Accuracy Using Better Set Allocation

As described in Section 5.2, set allocation manages the assignment of IG nodes to sets. Prior work proposed a greedy set allocation scheme which aggressively groups nodes together without limit. Due to the skewed allocation of instructions to sets, certain sets are overpopulated at runtime, whereas other sets remain under-utilized. A scheme that distributes instructions more uniformly across sets could provide better accuracy, and combined with our other approaches, may improve performance and accuracy together.

The insight for a better set allocation scheme arises from set customization. Given the selection described in Table 6.5, the set allocation could be tuned to create sets which only hold as many addresses as can be accurately supported by the underlying storage. For example, if we were to only use the 32-bit Single Integer Set, then since it can only hold a single element at any given time, the number of elements entered into such a set should be limited to 1. Hence, the set allocation scheme should create sets that work well with the underlying storage options available.

In our new set allocation scheme shown in Algorithm 10, PSets are seen as a bin with a fixed
size that can only support a max number of insertions, \textit{Max}. If a PSet becomes full, nothing else can be added to it. A node, \textit{N}, can be added to a PSet as long as doing so does not exceed result in its population exceeding \textit{Max}. New PSets can be created as needed to accommodate all queries. This approach limits overcrowding in sets by increasing the number of PSets.

Using this new set allocation, the impact on performance and accuracy is not easy to predict \textit{a priori} for a given application. By spreading insertions over a larger number of PSets, the cost of clearing PSets increases and the number of operations for conflict detection will likely increase though it is not guaranteed. At the same time, by spreading insertions over more sets, any

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_2}
\caption{Early Termination + Set Customization}
\end{figure}
individual disambiguation is less likely to result in a false positive.

**Algorithm 10: Population-based Set Allocation Heuristic**

**Input:** N: Candidate Node  
**Input:** P: Available PSets  
**Input:** Max: Max Weight allowed per Set

1. foreach $s \in P$ do  
2. if $\text{Population}(s) < \text{Max}$ and No edge from $s \to N$ then  
3. $s$.add(N);  
4. $\text{Population}(s) += \text{Population}(N);$  
5. break;  
6. end  
7. end

8. if N not added to any PSet then  
9. Make new set s;  
10. s.add(N);  
11. $\text{Population}(s) += \text{Population}(N);$  
12. end

There are a variety of techniques for computing the Population count for sets. We could use edge profiling or a static heuristic. We found a static heuristic that performed well, so we only present those results. Also, it has the advantage that additional edge profiling is unnecessary.

### 6.4.1 Static Insertion Counting

Static Insertion Counting treats all static insertions as having the same weight: 1. This scheme is simple, and performs surprisingly well. Given that the largest storage medium is 2K-bits, we restrict $\text{Max}$ to 20, since this is significantly less than $\frac{2000}{32}$ and will reduce the likelihood of overcrowding.

<table>
<thead>
<tr>
<th>Bench</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>[6, 10]</th>
<th>[11, 20]</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>450</td>
<td>10.5</td>
</tr>
<tr>
<td>swim</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mgrid</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>applu</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>33</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>vpr</td>
<td>0.5</td>
<td>1.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>167</td>
<td>3.7</td>
</tr>
<tr>
<td>mesq</td>
<td>2.7</td>
<td>0.3</td>
<td>0</td>
<td>0.3</td>
<td>0</td>
<td>2.3</td>
<td>66</td>
<td>3.3</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>2.3</td>
</tr>
<tr>
<td>ammp</td>
<td>0</td>
<td>1.9</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>0</td>
<td>27</td>
<td>115</td>
</tr>
<tr>
<td>parser</td>
<td>0.7</td>
<td>0.9</td>
<td>0</td>
<td>5</td>
<td>1.8</td>
<td>3.2</td>
<td>179</td>
<td>3.9</td>
</tr>
<tr>
<td>bzip2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>4.17</td>
</tr>
<tr>
<td>twolf</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>97</td>
<td>2.5</td>
</tr>
</tbody>
</table>
Table 6.8: System Configurations for new Population-Based Set Allocation.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocS</td>
<td>Baseline with Static Pop. Set Allocation</td>
</tr>
<tr>
<td>AllocS+ET</td>
<td>Alloc with Early Termination</td>
</tr>
<tr>
<td>AllocS+ET-S10</td>
<td>Alloc+ET sample at a rate of 10</td>
</tr>
<tr>
<td>AllocS+ET-S100</td>
<td>Alloc+ET sample at a rate of 100</td>
</tr>
<tr>
<td>AllocS+ET-S1000</td>
<td>Alloc+ET sample at a rate of 1000</td>
</tr>
<tr>
<td>AllocS+ET+C</td>
<td>Alloc+ET with Set Customization</td>
</tr>
<tr>
<td>AllocS+ET+C-S10</td>
<td>Alloc+ET+C sample at a rate of 10</td>
</tr>
<tr>
<td>AllocS+ET+C-S100</td>
<td>Alloc+ET+C sample at a rate of 100</td>
</tr>
</tbody>
</table>

Table 6.7 shows population counts using the static insertion counts along the columns as a percent increase over the previous greedy scheme. For sets sized 1 through 10, the percent increase is fairly modest. However, for sets sized from [11,20], the increase is substantial, varying from no increase to 450%. Finally, the last column shows the overall percent increase in total set count as compared to PSets Count in Table 6.1. Since the total number of PSets increases, we expect increase costs associated with Clearing and Conflict Detection.

The configurations evaluated are shown in Table 6.8. Combining this technique with early termination yields the performance and accuracy shown in Figures 6.3(a) and 6.3(b). Compared to Early Termination, this new design runs slower as expected. However, its accuracy is the same or better, on average, and for each benchmark.

The most interesting result, however, is that accuracy is not degraded during sampling at rate of 10, 100, or even 1000. Instead, accuracy is retained, no only on average, but for each benchmark. This tells a consistent story—careful attention to overcrowding in sets significantly boosts accuracy. Impressively, this design with a sample rate of 1000 achieves a \(2 \times\) slowdown with the same accuracy as the full run. Hence, we use more sets to increase accuracy but operate on them infrequently thanks to early termination.

Figure 6.4 shows the percentage of PSets and conflict checks for AllocS over Baseline, averaged for all benchmarks. The large number of new sets and conflict checks explain the significant slowdown of AllocS compared to Baseline and ET (shown in prior plots).

Figure 6.5(a) and 6.5(b) add the impact of early termination, set customization, and our new set allocation together. Interestingly, AllocS+ET+C gets a significant performance boost due to customized sets. When sampling is applied, the performance is reduced and is almost as good as that of Figure 6.2(a). However, the accuracy is significantly worsened due to the inaccuracy of the smaller sets across all three configurations.
Figure 6.3: Early Termination + Set Allocation Using Static Insertion Counting
### Table 6.9: Original Set Distribution

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>[6,10]</th>
<th>[11,20]</th>
<th>&gt;20</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>55</td>
<td>13</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>86</td>
</tr>
<tr>
<td>171.swim</td>
<td>16</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>173.aplu</td>
<td>98</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>129</td>
</tr>
<tr>
<td>175.vpr</td>
<td>230</td>
<td>68</td>
<td>15</td>
<td>29</td>
<td>8</td>
<td>16</td>
<td>9</td>
<td>7</td>
<td>382</td>
</tr>
<tr>
<td>177.mesa</td>
<td>1986</td>
<td>336</td>
<td>119</td>
<td>300</td>
<td>47</td>
<td>128</td>
<td>89</td>
<td>17</td>
<td>3022</td>
</tr>
<tr>
<td>179.art</td>
<td>25</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>43</td>
</tr>
<tr>
<td>188.ammp</td>
<td>274</td>
<td>107</td>
<td>135</td>
<td>11</td>
<td>8</td>
<td>22</td>
<td>13</td>
<td>11</td>
<td>581</td>
</tr>
<tr>
<td>197.parser</td>
<td>412</td>
<td>109</td>
<td>35</td>
<td>20</td>
<td>56</td>
<td>63</td>
<td>19</td>
<td>14</td>
<td>728</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>146</td>
<td>15</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>192</td>
</tr>
<tr>
<td>300.twolf</td>
<td>566</td>
<td>153</td>
<td>100</td>
<td>48</td>
<td>10</td>
<td>51</td>
<td>29</td>
<td>18</td>
<td>975</td>
</tr>
</tbody>
</table>

#### 6.4.2 Characterization

**Set Distribution**

Table 6.9 show the distribution of instructions across sets by population using the original set distribution scheme. Table 6.10 shows the same for the AllocS scheme. The AllocS scheme restricts the number of elements in each set to <20, hence there are no sets with >20 elements in them, in this scheme. Finally, Table 6.11 shows the change in set distribution from the original scheme to AllocS. We see that the application of AllocS not only effects the highest bracket [11 – 20], but also brackets leading upto 1. This is because of the way the Root Set allocation interacts with AllocS. Since there are a limited number of bins available for each instructions, due to the conflicts with other instructions, finding a bin is made even harder by AllocS. It is this change in set distribution that allows AllocS to scale performance to 2x while retaining high accuracy.

![Percentage of PSets and Conflict Checks compared to Baseline](image)

Figure 6.4: Percentage of PSets and Conflict Checks compared to Baseline
Table 6.10: AllocS Set Distribution

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>[6,10]</th>
<th>[11,20]</th>
<th>&gt;20</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>55</td>
<td>13</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>11</td>
<td>0</td>
<td>95</td>
</tr>
<tr>
<td>171.swim</td>
<td>16</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>173.applu</td>
<td>98</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>0</td>
<td>139</td>
</tr>
<tr>
<td>175.vpr</td>
<td>231</td>
<td>69</td>
<td>15</td>
<td>29</td>
<td>8</td>
<td>20</td>
<td>24</td>
<td>0</td>
<td>396</td>
</tr>
<tr>
<td>177.mesa</td>
<td>2040</td>
<td>337</td>
<td>119</td>
<td>301</td>
<td>47</td>
<td>131</td>
<td>148</td>
<td>0</td>
<td>3123</td>
</tr>
<tr>
<td>179.art</td>
<td>25</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>188.ammp</td>
<td>274</td>
<td>109</td>
<td>135</td>
<td>14</td>
<td>8</td>
<td>28</td>
<td>28</td>
<td>0</td>
<td>596</td>
</tr>
<tr>
<td>197.parser</td>
<td>415</td>
<td>110</td>
<td>35</td>
<td>21</td>
<td>57</td>
<td>65</td>
<td>53</td>
<td>0</td>
<td>756</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>146</td>
<td>15</td>
<td>11</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>9</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>300.twolf</td>
<td>566</td>
<td>156</td>
<td>103</td>
<td>49</td>
<td>10</td>
<td>58</td>
<td>57</td>
<td>0</td>
<td>999</td>
</tr>
</tbody>
</table>

Table 6.11: Percentage change in Set Distribution

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>[6,10]</th>
<th>[11,20]</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>450</td>
<td>10.47</td>
</tr>
<tr>
<td>171.swim</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>173.applu</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15.38</td>
<td>33.33</td>
<td>0</td>
<td>7.75</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.43</td>
<td>1.47</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>166.67</td>
<td>3.66</td>
</tr>
<tr>
<td>177.mesa</td>
<td>2.72</td>
<td>0.3</td>
<td>0</td>
<td>0.33</td>
<td>0</td>
<td>2.34</td>
<td>66.29</td>
<td>3.34</td>
</tr>
<tr>
<td>179.art</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33.33</td>
<td>2.33</td>
<td></td>
</tr>
<tr>
<td>188.ammp</td>
<td>0</td>
<td>1.87</td>
<td>0</td>
<td>27.27</td>
<td>0</td>
<td>27.27</td>
<td>115.38</td>
<td>2.58</td>
</tr>
<tr>
<td>197.parser</td>
<td>0.73</td>
<td>0.92</td>
<td>0</td>
<td>0</td>
<td>1.79</td>
<td>3.17</td>
<td>178.95</td>
<td>3.85</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>11.11</td>
<td>0</td>
<td>4.17</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0</td>
<td>1.96</td>
<td>3</td>
<td>2.08</td>
<td>0</td>
<td>13.73</td>
<td>96.55</td>
<td>2.46</td>
</tr>
</tbody>
</table>
Membership Counts

Figure 6.6 shows the impact of early termination of set allocation schemes, Basic and AllocS. The insertion counts are not impacted by the set allocation schemes, as each instruction only inserts into a single set. Membership counts are a function of the number of sets that need to be checked on average, at each instruction. Early termination removes redundant checks on-the-fly. We see significant performance improvements on application of ET, and this graph explains the
reason for the change. In a non-sampled run, ET decreases membership counts to 14% of their original value, i.e. it decreases the count by almost a factor of 10. Besides the primary impact of reduction in actual instructions to be executed, this also decreases the impact of the signature profiler on the cache, as membership checks in the SCM queries involve checks with sets used at the beginning of the region. A combination of these effects is responsible for this improvement.

Figures 6.7 and 6.8 show the fractional decreases in insertion and membership counts brought about by sampling. Figure 6.7 shows the trend for the basic set allocation scheme, whereas Figure 6.8 shows the same for AllocS.

6.5 Evaluation

6.5.1 Comparison with Prior Set Profiler Configurations

We adopt AllocS+ET-S100 and AllocS+ET-S1000 as our new preferred profiling schemes and compare them to BestSig and FastSig from Chapter 5. The BestSig configuration here is a sampled version of the AccuSig profiler, whereas the FastSig is the fastest configuration obtained using 8K single array bloom filters. These configurations are compared against the AllocS+ET-S100 and AllocS+ET-S1000, the two fastest configurations presented in Figures 6.9(a) and 6.9(b) respectively. The two new configurations are clearly faster than both prior configurations, while closely matching the accuracy of BestSig. The slowest benchmark in BestSig is ammp, which is slower than the baseline by 130x. The same benchmark is 2x slower than the same
Figure 6.7: Fractional Decreases in Insertion and Membership Counts With Sampling
Figure 6.8: Fractional Decreases in Insertion and Membership Counts With Sampling
baseline using AllocS+ET-S1000, while matching the accuracy. Together, ET and AllocS allow us to achieve much better trade-offs between performance and accuracy.

Next, we look at the impact of accuracy on a speculative LICM pass. The profile data is fed back into the compiler, and through the alias analysis, LICM is allowed to hoist loads and dependent instructions speculatively. The percentage difference between the loads and instructions hoisted by the ideal versus a couple of AllocS-ET configurations is shown in Figures 6.10(a) and 6.10(b). Our results are a couple of percent lower overall, compared to prior work, and we attribute this to the slight deterioration in accuracy (NAED 0.16 to 0.19). The difference in percentage of zeroes identified (not shown in the figure) is 2.5%. This compares to <1% for

![Graph showing performance and accuracy comparison]

(a) Performance

(b) Accuracy

Figure 6.9: Comparison with Prior Configurations

77
the initial version presented in Chapter 5. Overall, we achieve a 4x performance improvement over prior work while closely matching accuracy. This is achieved through a combination of techniques which provide better set allocation and eliminate redundant checks on-the-fly.

6.5.2 Improving Traditional Profilers using Sets

Figures 6.11, 6.12 and 6.13 show the performance and accuracy when applying the set creation and assignment, described above, to traditional profilers. The traditional profiler here still uses centralized storage, but instead of using it to simply store the occurrence of a previous instruction, the set(s) associated with the prior instruction(s) is(are) stored.

Each instruction in the set-based profiler is allocated a set. The set-based profiler inserts this instruction into the given set. In this traditional profiler configuration, the set number is stored at the entry indexed using the instruction’s address. This allows the instruction to distinguish between prior instructions which come from different sets. Therefore, using static

![Graph](image-url)

(a) Percentage of Loads Hoisted vis-a-vis Ideal

![Graph](image-url)

(b) Percentage of Instructions Hoisted vis-a-vis Ideal

Figure 6.10: Effect on Speculative LICM
analysis, each instruction at runtime only needs to check against a given number of sets. In this case, the instruction would check if any of the bits associated with those sets are marked in the hash table entry. If there are prior instructions that belong to different sets, that use the same

![Bar chart](image)

(a) Performance

![Bar chart](image)

(b) Accuracy

Figure 6.11: Traditional Profilers using Sets
entry, this will not affect the accuracy of this instruction.

In addition to using sets to find relevant prior instructions, we also incorporate early termination, removing redundant checks on-the-fly. This combination allows us to significantly improve

Figure 6.12: Traditional Profilers using Sets with Sample Rate 10
the performance of the traditional profilers while simultaneously improving their accuracy. The figures below quantify these benefits.

The traditional profiler is able to profile the benchmarks with an accuracy of 0.33 NAED

![Bar charts showing performance and accuracy of traditional profilers using sets with sample rate 100.](image)

(a) Performance

(b) Accuracy

Figure 6.13: Traditional Profilers using Sets with Sample Rate 100
with a slowdown of 2.75x versus the baseline (Figure 6.13). The most accurate configuration has a slowdown of 18.24x with an accuracy of 0.31 NAED (Figure 6.11).

Clearly, set-based techniques, and the optimizations discussed in this chapter can be applied to traditional profilers to improve their performance and accuracy. However, the set-based profiler still outperforms the traditional profiler in both performance and accuracy. This is because the set-based profiler embeds the information gathered from static analysis in the storage scheme itself, creating distributed storage. This allows the accurate sets to be isolated from the inaccurate ones, and affords the use of signatures which provide quick and accurate operations.

### 6.6 Summary

An initial version of the profiler was able to profile programs from the SPEC benchmark suite with a slowdown of 8x, with good accuracy (0.15NAED). This version of the profiler relied on a hill-climb based design space exploration to achieve this performance and accuracy point. Our evaluations in this phase showed that the profiler could traverse a large space of performance and accuracy tradeoffs. More importantly, the profiler’s behavior was predictable once the programs, and the set profiler’s knobs were understood. This knowledge allowed us to further optimize the compiler with novel optimizations.

*Early Termination* This optimization allows the profile to detect redundant checks on-the-fly and disable them, hence decreasing overhead without affecting accuracy. We found that this technique helps traditional profilers improve their performance without a loss of accuracy. This technique was able to speed up the profiler by a factor of 2, against the BestSig configuration with 8x slowdown.

*Edge Profile Set Sizing* This optimization tuned each individual set used by the profiler, according to its population. This tuning allowed us to use different types of signatures for different sets within the same profile run. This optimization, when coupled with Early Termination, sped up the profiler by 4x, against the BestSig configuration, however there was a significant deterioration in accuracy.

*AllocS Set Allocation* The basic and Root Sets set allocation schemes assign sets based on the interferences in the dependence graph. In order to make set allocation schemes aware of the population counts within, we introduced the AllocS set allocation. In addition to the Root Sets allocation, this scheme limited the number of instructions entering a single set. This allowed the sets to maintain accuracy even when their sizes were decreased. Coupled with the Early Termination, this configuration improved performance when compared to BestSig, by a factor of 4, with a negligible loss in accuracy.
Chapter 7

Feeding back Data Dependence Profile Information

This chapter explores the challenges and opportunities in feeding back data dependence profile information to the compiler.

In order to better leverage the available data dependence information, we first perform edge profile feedback, which can simplify control flow, and expose greater opportunities for the data dependence profile that follows.

In all the feedback mechanisms described here, we assume a zero cost recovery scheme to help maintain semantics. The goal of this chapter is to establish the potential for speculative optimizations utilizing data dependence profile information. This potential is quantified as speedups over non-speculatively optimized code. Using PTLSim [38], a cycle-accurate x86 simulator, the sensitivity of the speculatively optimized code to various processor configurations is explored.

Initially, a description of different types of profiles and their roles in optimizing the compilers is provided. Here, we describe the role that is played by edge profile in improving the quality of code for the subsequent data dependence profile.

Later, the results of the feedback system are presented. As mentioned in Chapter 3, this feedback system is implemented as a standalone tool which is compatible with LLVM 3.1. PTLSim is the chosen evaluation platform. Multiple processor configurations are tested in order to identify the interaction between FDO and the processor architecture.

7.1 Feeding back profile information

Broadly, there are three different types of profiles that can be gathered from a program. They are:
• Edge Profile provides control-flow information

• Data Dependence Profile provides memory-carried dependence information

• Value Profile provides runtime values at a given program state

Here, mechanisms for feeding back edge and data dependence profile information are presented. Edge profile allows the compiler to remove unused blocks from the code, and straighten and simplify the control-flow in the program. As the studies will corroborate, this has a significant impact on performance of the programs. Performing a data dependence profile subsequently helps the compiler perform better scheduling and eliminate redundant instructions. The edge profile feedback mechanism is described next.

7.1.1 Running Example

Figure 7.1 shows an example of code that will be used to show the effects of profile information on conventional codes. The example shows a CFG with six basic blocks, that forms a function. Each block’s name is shown at its top-right corner. The first block is the Entry block. Only the memory operations, and branch instructions are shown. This is because edge and data dependence profiles (the two types of profiling that will be explored in the subsequent sections)
directly impact these instructions. Each instruction consists of <opcode> <operand> <optional operand>. Using this notation, the instruction types shown are:

- LD <operand>: Load instruction accessing memory location given by <operand>
- ST <operand>: Store instruction accessing memory location given by <operand>
- BR <true_cond> <false_cond>: Branch instruction with true and false targets.
- BR <uncond_target>: Branch instruction with a single target

It is assumed that the alias analysis was unable to ascertain any specific relationship between the memory operations shown here. In other words, the alias analysis’ response to a query consisting of any two memory operands in this example, would be a may_alias. In the process of profiling, these relationships are quantified, creating opportunities for optimization.

In the next section, edge profile information is used to optimize the code. Subsequently, data dependence profile finds further opportunities resulting in a drastically simplified code.

### 7.1.2 Edge Profile Feedback

An edge profile is designed to detect hot portions of code within a program. This information is useful in determining the choices made by optimizations like PRE, which tries to remove redundant code along hot paths, often resulting in extra code on the cold path. These optimizations are trying to decrease the length of the hot path, at the expense of the cold path. This category of optimizations require runtime information (or static analysis) to recognize hot regions of code. Edge profile information provides this information.

Prior works have proposed multiple techniques for collecting an edge profile efficiently. Given the limited amount of information that needs to be collected, even a very basic technique for collecting the profile (like the unoptimized version of the edge profile in LLVM 3.1) is able to quickly gather control flow data.

In order to explore the speculative optimization potential using an edge profile, we follow the steps listed below.

- Collect edge profile information for given program using training input
- Eliminate all basic blocks that are unused during the profile run (Dead Block Elimination)
- Aggressively optimize the resulting code
- Execute speculatively optimized code on training input (same input as above)
Figure 7.2: An example code

Figure 7.3: Impact on code

Figure 7.4: Applying basic block straightening impacted code
The speedup is measured as the ratio of the execution time of speculatively optimized code versus a non-speculatively optimized version. Given this background, let’s look at how edge profile can affect the running example.

Figure 7.2 shows the *hot edges* in the running example. Hot edges refer to edges that were actually active during the profile run. In other words, the blocks listed here are the only ones that were exercised (touched) during the profile run. In the current scheme, the other blocks can be eliminated. The effect of eliminating the blocks is shown in Figure 7.3. As block 2 was not touched, Dead Block Elimination is able to remove it from the CFG. This simplifies the branch instructions in surrounding blocks. Entry block now has a single successor, hence the conditional branch is now converted to an unconditional one. The Exit block now has a single predecessor. Both these cases provide opportunities for further opportunity. Basic block straightening can now merge block 3 with Entry, and block 5 with Exit. The resulting code is shown in Figure 7.4. There are two important points to note regarding this change.

**Enabling other optimizations** The change was brought about by a classical compiler optimization (basic block straightening) which is part of most optimization levels. The speculative change that was performed by Dead Block Elimination created further opportunities for optimizations. At a larger scale, these changes can create a domino effect where passes now find more sites to transform. A significant part of the potential for profile-directed optimizations lies in the ability of the provided information to enable more optimizations, beyond the first set. This will play out in a significant way in the next section when data dependence profiling information is introduced.

**Limits to optimization** Comparing Figures 7.3 and 7.4, we notice that the memory operations have not been re-ordered. This is a limitation of the type of profile information provided. Since the compiler does not have any more information regarding memory operands than it did earlier, it is unable to perform any further optimization. Without collecting the necessary data dependence profile, the performance potential of the edge profile feedback is limited. Conversely, block 2 contained memory operations which need not be profiled, since they are not in a hot block. Hence, the edge profile decreased the scope of the data dependence profile. This decreases the profiling waste, and reduces the challenges in feeding back data dependence profile information to the compiler. For this reason, in our studies, the edge profile is performed prior to the data dependence profile.

This inter-dependence between profiles is an interesting insight of the study that was performed. It makes the case for gathering more runtime information, in order to fully realize the optimization potential. Next, a data dependence profile and its effect on the code are described.
7.1.3 Data Dependence Profile

Data dependence profiling was introduced in Chapter 2, and explained in subsequent chapters. Here, the impact of data dependence profiling on speculative optimizations is explored. More precisely, we try to provide a high-level view of the types of optimizations that are likely to benefit from a given type of profile, and how this impacts program performance.

Broadly speaking, a data dependence profile tries to ascertain the presence or absence of dependences, with a high enough probability. The probability at which the probability information becomes useful, depends on the speculative optimization system. In the current system, we define the presence or absence of a dependence (sometimes called the threshold) as the following.

- A dependence exists if the two values always hold the same value. (Probability: 1)
- A dependence does not exist if the two values never hold the same value. (Probability: 0)

Hence, we are restricted by the system to only those dependences that never occur (absence of a dependence), or that always occur (presence of a dependence). This limitation implies that, on the same input, the code may be speculatively optimized using this data dependence profile information, without the need for recovery.

We’ll explain the effect of data dependence profile on the code already optimized by edge profile, in the next couple of sections.

Absence of dependence

From the data dependence profile provided, we extract those dependences that never occurred. These dependences have an alias probability of zero, (i.e. no dependence was recorded during the execution run). This information is used by optimizations to re-order memory operations. Unlike control flow information, which is available in the structure of the code, data dependence information is provided in compilers through alias analyses. This design allows multiple optimizations which require this information, to receive it reliably. From a software engineering perspective, a single alias analysis interface simplifies code design both for optimization writers and alias analyses writers.

Therefore, in order to leverage the no-alias information, it is made available to optimizations through the alias analysis. Optimizations like LICM which perform code scheduling use this information to reorder instructions. In the running example, the profile information regarding the absence of dependences is shown in Figure 7.5. Using this information, a global instruction scheduling pass is able to reorder the memory operations. The resulting code is shown in Figure 7.6. Notice that since memory operands E and F don’t alias with D and A, and C doesn’t alias with A, all the loads are hoisted above the stores. This can be performed despite
the control flow, since each of the loads is guaranteed to happen, and moving them above the branches doesn’t change program semantics, as long the profile information is relevant.

**Code Scheduling** In our studies, we found that effect of no-alias information is seen mostly in code scheduling. Optimizations which can reduce instruction count, or improve instruction quality, through code scheduling appear to benefit from this information. Feeding this information back to superscalar processors, we found that the impact is small, as the dynamic, out-of-order scheduling structures already perform good scheduling within a given window. LICM is an optimization that was able to move instructions out of the loop using this information. This transformation decreased the number of instructions that needed to be executed. Although this was not a common case, it shows that in certain cases, this information can lead to significant payoffs.

**Presence of dependence**

Data dependence profiles can also ascertain that a certain dependence occurred on each encounter during the profile run. To understand the utility of this information, its important to understand the semantic meaning that is ascertained with the presence of a dependence. Specifying that two values (addresses) in the program are the same suggests redundancies within the program. The same value is being generated by different control- and/or data-flow paths. In effect, replacing one value with the other wouldn’t change the semantics of the program. Combined with static single assignment (SSA), this information becomes more potent, as all the instances of one value can be replaced by another. Since the generation of values within the program could include expensive instructions like loads and stores, removing a definition’s uses in the program would
cause its generating slice to be declared dead. This behavior can have a cascading effect. Our studies show that this is indeed the case in atleast some of the benchmarks.

Let’s take a look at the effect that this information can have the already optimized example code. Figure 7.7 shows the must aliases that were reported by the data dependence profiler. In this code, the profiler was able to determine that addresses E, F and C are all equal. In addition, (optimistically) its able to prove that A and D are also equal. This information can decrease the number of loads to 1, and replace both the stores with one store in the return block. These optimizations are enabled by Redundant Load Elimination and Dead Store Elimination. Both these optimizations are part of traditional compiler frameworks. As the information is made available through the alias analyses, no additional passes are necessary to allow the compiler to leverage this information.

Removing Redundancies As mentioned previously, must-alias information is powerful as it points to locations where redundancies exist. In constrast to code scheduling, redundancy removal is more potent as it often leads to removal of instructions. This decreases the number of instructions that need to be executed. Hardware does not have this luxury as it is forced to execute each and every instruction provided in the binary.

We evaluate the effect of edge and data dependence profiling on the SPEC benchmark suite in the next section. In the example discussed, a combination of the profiles was able to significantly simplify the code.

7.2 Evaluation

To explore the potential for profile-directed optimizations, we feed back the edge and data dependence profiles to the LLVM compiler at IR level. The edge and data dependence profilers were implemented as a standalone tool, which is compatible with LLVM 3.1. The workflow
Table 7.1: Processor Configurations

<table>
<thead>
<tr>
<th>Specification</th>
<th>Default</th>
<th>Config 2</th>
<th>Config 3</th>
<th>Config 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Queue</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Max Issue Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Phys. Reg. File</td>
<td>256</td>
<td>64</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>ROB Size</td>
<td>128</td>
<td>32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>In-flight Branches</td>
<td>16</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Load Queue</td>
<td>48</td>
<td>12</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>Store Queue</td>
<td>32</td>
<td>8</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Fetch Queue</td>
<td>32</td>
<td>8</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Fetch Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Frontend Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Frontend Stages</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Dispatch Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Writeback Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Commit Width</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Forwarding Latency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Gzip IPC</td>
<td>1.1</td>
<td>0.6</td>
<td>0.93</td>
<td>0.82</td>
</tr>
</tbody>
</table>

for the tool was presented in Figure 3.4, and its reproduced here as Figure 7.9 for readability purposes.

7.2.1 Setup

Using the results of the edge profile, unused blocks from the code are removed. Subsequently, the set-based profiler marks all eligible data dependence edges for profiling. Unlike prior chapters, where SCM queries were used for profiling, here we instrument all may-alias edges in the program. The profiler identifies the no-aliases during the profile run. This information is fed back into the compiler for speculatively optimizing the code.

**Must-Alias Profiler** In addition to the no-alias profiler, we also instrument the code to find a limited number of must-aliases. The current implementation of this profiler is basic, and is intended to find common addresses across loads and stores in the program. This information is then used to enable other optimizations find redundancies within the code. Specifically, a speculative optimization pass uses the must-alias information to replace two must-alias operands, with the dominating production. This allows later passes like dead code elimination (DCE) and partial redundancy elimination (PRE) to identify and remove any redundant/dead code. In other words, this profile tries to identify redundant address generation slices in the program.

**Processor Configurations** The speculatively optimized code is executed on the same profiling input, and hence, doesn’t require recovery code. In order to test the sensitivity of the optimized code to architecture parameters, we test the code on multiple architectures. These
Figure 7.9: Workflow with DDP and Edge Profile feedback to FDO
Table 7.2: Evaluated Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Baseline configuration using O3 optimizations and no profile information</td>
</tr>
<tr>
<td>E</td>
<td>Feedback configuration using Edge Profile</td>
</tr>
<tr>
<td>EN</td>
<td>Feedback configuration using Edge and No-Alias Data Dependence Profile</td>
</tr>
<tr>
<td>ENM</td>
<td>Feedback configuration using Edge, No-Alias and Must-Alias Data Dependence Profile</td>
</tr>
<tr>
<td>x-S</td>
<td>Configuration $x$ using Set-based Profiler information</td>
</tr>
<tr>
<td>x-I</td>
<td>Configuration $x$ using Ideal Profiler information</td>
</tr>
<tr>
<td>x-T</td>
<td>Configuration $x$ using Traditional Profiler information</td>
</tr>
</tbody>
</table>

itectures are listed in Table 7.1. It is reproduced here as Table ???. All the configurations shown here use out-of-order superscalar processors. The changes in parameters determine how aggressively the processor is able to find independent instructions to execute out-of-order. To rank the processors in order of performance, the IPC from the benchmark 164.gzip is shown in the line of the figure. The default configuration is the most aggressive configuration tested, whereas Config 2 is the least aggressive. Config 3 and 4 span the interval between the prior two configurations. Using these configurations also enables us to understand the interaction between the optimizations the processor parameters.

**Metrics** The baseline for the runs is an aggressively optimized non-speculatively optimized version of the benchmark. Three main metrics are collected for each run, to compare the performance, namely, IPC, cycle count and number of instructions. The regions of simulation chosen here are provided by SESC simulation marks. We have ported these marks to the PTLSim simulator, to simulate representative regions of code. Unlike SESC however, the benchmark is run natively prior to these regions. Hence, the caches and branch predictors are not warm. Native execution of the un-simulated regions of the benchmark enables quick simulations.

We first present the speedup obtained by feeding back edge profile information into the compiler. Subsequently, we show the results for feeding back DDP information.

**Configurations Tested** The baseline and feedback configurations are listed in Table 7.2. The baseline configuration uses no feedback information, and is an aggressively optimized non-speculative version of the code. The baseline is compiled using the same LLVM 3.1 infrastructure used by the feedback framework. Configuration E uses edge profile information in the feedback framework. Configuration EN uses edge and no-alias data dependence profiles. The underlying profiler for the data dependence profiler can be a set-based profiler (S), ideal profiler (I) or a traditional profiler (T). The most speculative configuration is ENM, which uses edge, no-alias and must-alias data dependence profile information to speculatively optimize the code.
7.2.2 Edge Profile

Figures 7.10 and 7.11 show the improvement in performance achieved with the help of edge profile information feedback. Figure 7.10 shows the improvement in IPC measured as a ratio with the baseline IPC. The average is the geometric mean of the values of each benchmark. Figure 7.11 shows the percentage improvements in cycles and instructions over the baseline. The average is measured as the arithmetic mean percentage improvement for each benchmark.

The edge profile information is fed into a dead block elimination removes unused blocks from the code, and subsequent passes simplify the code leveraging the simpler control-flow within the program. These transformations typically decrease instruction counts, or provide better instructions, resulting in lesser cycle counts. Since instruction count varies from the baseline to the profile-optimized run, due to the speculative optimizations, cycle counts and instruction decreases are the important metrics which explain the impact of the feedback information.

Instructions In terms of changes in instructions, 164.gzip, 197.parser and 168.wupwise show significant improvements. Reduction in instruction counts in these cases can be attributed to better code generation resulting from the removal of unused blocks. The instruction count is determined by how well the compiler was able to optimize the code, irrespective of the architecture. Here, dead block elimination only removes the un-used blocks, which do not directly contribute to any instructions in the dynamic execution of the program. Hence, the effect that we see here is the domino effect of dead block elimination on other optimizations in middle-end and backend of compilers. In 179.art it appears that the removal of unused blocks led to slightly
Figure 7.11: Percentage Improvement using Config E
worser code generation, possibly due to more aggressive inlining, and the subsequent register pressure resulting in more spills and fills.

**Cycles** This metric shows the improvements in execution time afforded by the speculatively optimized code. The average improvement in cycles closely matches the instruction improvement number, which suggests that the instruction mix, on average, has not radically changed from the baseline. However, certain benchmarks see significant improvements in cycles, close to 15% in the case of wupwise. 256.bzip2 presents an interesting case where the number of cycles has decreased despite stagnant instruction counts. This points to better instructions in the binary. In art’s default configuration run, the deterioration in instruction count is compensated to some extent by instructions that execute faster, hence improving IPC. The fact that this is not seen across other configurations for art, shows that the other processors are unable to make use of this better mix of instructions, possibly due to their less aggressive setups. Wupwise sees the highest improvement in cycles in the most conservative configuration, Config 2. This is possibly because the baseline performance for this configuration was probably poor. Speculative optimizations have enabled a more conservative architecture to post significant improvements in performance. Swim and, to a smaller extent, parser highlight this point as well.

### 7.2.3 Edge and Data Dependence Profile

Here, we feed back the data dependence profile information, in addition to the edge profile information fed in earlier. The goal is to add to the direct control-flow optimization, with direct data-flow optimizations. Both these techniques have secondary effects in the control- and data-flow, as they enable other optimizations.

**No-Alias Feedback**

To begin with, we add no-alias data dependence profiling information and measure its impact on performance. Figures 7.12 and 7.13 show the impact of no-alias data dependence information on SPEC2000 benchmarks. The new data only has measurable impact on performance in three benchmarks - 300.twolf, 183.equake and 179.art. Surprisingly, 183.equake shows no improvement with edge profile feedback, and 179.art, in fact, deteriorates due to poorer code generation. The addition of no-alias information allows optimizations like LICM to find redundancies within these benchmarks, and deliver performance improvements. In 183.equake, LICM aggressively hoisted 10x the instructions hoisted in plain profile, and 9x the number in edge profile.

The results seem to point to the fact that utility of no-alias information could be limited to specific types of benchmarks, where the redundancy could be captured with this type of information. It could also be the case that adding other types of profile information, like value profile or must-alias profile could remedy this situation. The encouraging point from these
results is that there is untapped performance in certain types of benchmarks that can be realized with no-alias information.

**Must-Alias Feedback**

Figures 7.14 and 7.15 show the impact of data dependence profile information (no-alias and must-alias) over the edge profile feedback. We see that besides the three benchmarks - 300.twolf, 183.equake, 179.art - that realize performance improvements from the no-alias information, 164.gzip and to a lesser extent, 256.bzip2, realize performance gains. The performance improvement in 164.gzip is especially surprising since the instruction count does not change significantly, but the cycle count decreases significantly. This corroborates the increase in IPC for this benchmark. This appears to point to the fact that the instruction mix has improved.

Figure 7.16 shows the overall improvement in performance due to profile feedback, compared to the baseline. On average, there is a 5.4% improvement in cycles, and a 4.4% improvement in instructions. These improvements are dominated by a few benchmarks, which show significant benefits from the control and data dependence feedback. It is interesting to note that in some cases, the benchmark either solely benefited from the control flow information, or data dependence information, but not from both. We hypothesize that this points to the type of opportunities available in the benchmark.
Figure 7.13: Percentage Improvement using Config EN-S
7.2.4 Comparison with Traditional Profilers

Figure 7.17 shows the benefits from using the set-based profiler information, instead of traditional profilers. Here, we show the normalized IPC, and percentage improvement in cycles and instructions when using set-based profiler information, instead of traditional profiler information. We see that the benefits in quake and bzip2 are realized due to the improved accuracy afforded by the set-based profiler. This figure clearly shows the impact of accuracy on optimization potential. The decentralization of storage into various sets, allows the set profiler to isolate potentially inaccurate regions of code into a finite number of sets. As we mentioned before, the Root Sets and membership checks allow the set profiler to create highly local sets. An advantage of these localized sets, besides better cache locality, is the isolation from the rest of the storage. In other words, even if a few sets sets are completely full, and hence lose accuracy, the other sets are not impacted by this degradation. Our set-allocation schemes improve the accuracy of the traditional profiler by using the storage to record set IDs of the previous instructions, rather than simply setting a bit (Streamlined Design), or storing a tag (Improved Original Design). However, this improvement cannot get around the fundamental roadblock in traditional profiler design, that of centralized storage.

The results highlight how the difference in the percentage of zeroes that were profiled accurately, can impact the final optimization performance. As we showed in prior chapters, the set profiler is able to identify most of the zeroes in these profiles correctly. The same doesn’t hold true for traditional profilers, which on average captured only 66% of the zeroes on offer.
Figure 7.15: Percentage improvement using Config ENM-S instead of E
Figure 7.16: Performance improvement using Config ENM-S over B
Figure 7.17: Performance improvement when using Config ENM-S instead of ENM-T
As we see in the next section, since the set profiler is able to capture almost all of the zeroes correctly, it matches the ideal profiler in terms of the optimization performance.

### 7.2.5 Comparison with Ideal Profiler

Figure 7.18 shows the difference in performance between the set-based profiler and the ideal profiler. Given the high accuracy numbers of the set-based profiler, it is not surprising that the set-based profiler closely matches the ideal profiler in performance. Please note that the scale here is much smaller than the previous figure, to highlight the small changes. In equake for example, the set-based profiler found as much opportunity as the ideal profiler. A similar trend is seen in other benchmarks. The ideal profiler ran out of memory while profiling parser, wupwise and ammp, and hence we do have numbers for those benchmarks here.

Figures 7.17 and 7.18 clearly show that accuracy matters when it comes to optimizations. Even with the limited optimization potential seen in the benchmarks shown here, we see performance difference in programs, which can be attributed to their accuracy numbers. This also shows that NAED is a good metric for gauging the accuracy of a given profile.

These results clearly show that better accuracy unlocks more performance within programs, by decreasing the number of dependence constraints.

### 7.3 Summary

In order to demonstrate that there exists untapped performance in programs, and that it can be realized through profile information, we built a feedback-directed optimization framework using a state-of-the-art compiler. The framework is a standalone tool, which is compatible with LLVM-3.1. The framework includes the following profilers.

- A global edge profile

- A data dependence profiler which can profile
  - No-aliases using the set-based profiler
  - Must-aliases using a simple value-checking mechanism

The edge profiler is a pre-conditioning pass, which simplifies the control flow. This pass is performed in order to decrease the profiling load for the profilers, and improve the optimization potential for the optimization framework.

In addition to these profilers, the framework also includes a few passes which directly utilize the profile information to transform the code. These passes represent examples for further speculative optimizations which can harness the full potential of the profile information provided.
Figure 7.18: Performance improvement using Config EN-I over EN-S
Since we are trying to explore the potential for optimization, the tool assumes zero cost recovery. In our tests, we use the same input for profiling as well as execution. This obviates the need for a recovery mechanism.

Our results show that there is significant performance that can be had by unlocking conservative constraints in programs. At a high-level, the performance numbers represent the impact of writing code which can handle multiple inputs. This over-provisioning of the code tends to hinder compilers from performing optimizations, in order to maintain semantic correctness under all inputs. We relax such constraints by restricting the input set to the profiling input.

The profile information is collected from set-based, ideal and traditional profilers. This information is fed back into the framework, and their effect on the optimization potential is quantified. We find that there exists optimization potential within certain benchmarks, and that this potential can be tapped using accurate profile information. The inaccuracies in traditional profilers hinder their ability to utilize this potential. On the other hand, the set-based profiler is able to identify almost all of the potential (as measured by the ideal profiler) due to its accurate profiles.

Using the set-based profiler scheme, a 4.4% improvement in dynamic instruction count is realized, leading to a 5% improvement in execution time. In equake, we see a 20% improvement in instruction count, leading to a >10% improvement in cycles across all configurations. Further experiments showed that these results match the opportunity found using an ideal profiler. This shows that the set-based profiler is accurate enough to realize almost all of the opportunity available. This is in sharp contrast to the traditional profilers which are unable to realize these gains due to the inaccuracies in their profiles.

**Future directions** The framework shown here profiles every single may-alias dependence in the program, and combines that with the edge profile information to highlight the opportunity for speculative optimizations. There are two dimensions that could be explored in future studies.

The first dimension would be value profiling, which could be useful in removing other redundancies. Although, value profiling might seem to be a small addition, each additional piece of information can enable opportunities that were hitherto hidden. As an example, edge profiling enables better use of data dependence profile information, by reducing the control flow hurdles for optimizations. Incorporating the value profile could improve the effectiveness of the data dependence profile information as well.

The second dimension is the type of speculative optimizations that can be used to leverage the profile information. We have restricted the search here to mostly traditional optimizations which leverage the alias analysis to perform optimizations. They were fed speculative information through this interface in order to perform risky transformations. This approach requires lesser effort on the part of the programmer, but limits the types of transformations that are performed. Optimizations designed to leverage profile information could greatly improve the
impact of the profile information. One such example is the redundant block elimination pass, which wasn’t present in LLVM. The pass uses edge profile information to remove dead blocks. This optimization is able to greatly simplify the code.
Chapter 8

Related Work

8.1 Data Dependence Profiling

Profiling and speculation frameworks have been implemented in several previous works [20, 21, 9, 10, 28, 7, 17], however most approaches have focused on a particular class of applications, rather than a unified approach to the diverse range of applications requiring such support. In [9], Chen et al. propose a data dependence profiler targeted at speculative optimizations. They perform speculative PRE and code scheduling, using a naive profiler and speculative support provided through the ALAT (Advanced Load Address Table) in the Itanium® [22] processors. Their non-sampling profiler has a slowdown of up to 100x, and they propose sampling techniques to overcome this problem. Here, we propose a set-based profiler which can profile at high speed with good accuracy.

In [32], Wu et al. use the concept of independence windows and dependence clustering to find opportunities for speculative parallelization. The proposed profiler, called DProf, performs iteration-grain disambiguation and differentiates between intra- and inter- iteration dependences. This is a form of optimization-driven profiling, but they target a more specific and limited set of relationships.

Several static techniques providing data dependence information have been proposed. In [14], Hinds et al. show that the optimizations only query a small sub-set of the universal set of instruction pairs. Therefore, profiling by the needs of the optimizations provides a more scalable approach. In [28], Silva et al present a static analysis which computes pair-wise probabilities for the entire program. In addition to being inaccurate, the approach involves waste since [14] shows that not all pairs of instructions are relevant to optimizations. Conventional pointer analyses do not provide probability information for dependences. Chen et al. in [8] propose probabilistic points-to analysis algorithm using an interactive data-flow framework. Their model makes the conservative assumption that the heap is a single location. Furthermore, they use
Emami’s algorithm [11], which is not scalable to large programs.

In [32], Wu et al. use the concept of independence windows and dependence clustering to find opportunities for speculative parallelization. The proposed profiler, called DProf, performs iteration-grain disambiguation and differentiates between intra- and inter-iteration dependences. This is a form of optimization-driven profiling. Kim et al. in [17] propose a data dependence profiler that tries to identify opportunities for parallelism. Yu et al. in [36] propose a profiler which specifically targets loops for reordering transformations and parallelism. The profiler only profiles a partial dependence graph which is deemed sufficient for providing the information for loop reordering and transformation. The work is split into slices which can be executed in parallel, and combined into the partial dependence graph. Our profiler tries to find the alias in a given region, and could be extended to work with loops.

Recently, Yu et al. [37] proposed a DDP that produces slices that are then profiled on different cores. Here, we take an optimization-driven approach and only profile the relevant queries. Even so, we profile >80% of the IR-level memory operations in the benchmarks on average.

8.2 Speculative Optimizations

Roy and Srikant in [26] describe a modified SSA form which incorporates hotness of paths in SSA itself. In this representation, only definitions along the hot path reach their uses. The effectiveness of this scheme, referred to as Hot Path SSA, is evaluated on statistics gathered from a sparse conditional constant propagation pass, based on Wegman and Zedack’s algorithm from [31].

Kelsey et al. proposed a software speculation mechanism called Fast Track in [16]. This mechanism offloads semantic correctness checking away from the main thread. The speculative thread performs risky calculations, while the parallel thread performs runtime checks and parallel execution to ensure correctness. The applications for this scheme include profiling and speculative optimizations. In evaluations, the scheme is used for parallelizing hot loops in SPEC benchmarks. The technique is not designed to be a completely automatic process, and hence, requires user input.

In [33], Xue et al. propose a computational and lifetime optimal algorithm for PRE using execution profile and SSA. The main contribution of the paper is the algorithm that performs optimal speculative code motion using the SSA form. The authors show that it is possible to build flow networks out of the SSA graphs, and use the min-cut to find optimal code placement. The work is implemented in the Path64 compiler [2], and evaluated on SPEC benchmarks. They realize an average speedup of 2.76% compared with prior schemes.

In [23], Naveen et al. propose a scheme that uses hardware atomicity to handle control
speculation. Their scheme uses atomic regions to definition speculation boundaries, and discards work when certain invariants for guaranteeing correctness do not hold. The scheme is evaluated in a JVM, with aggressive inlining. Results show that combining atomicity and inlining has disproportionate impact on the code, possibly exploiting other optimization opportunities.

In [13], Gupta et al. describe various types of data dependence profiles and their possible uses in optimizations. The publication also briefly describes how such changes might impact compiler-directed architectures like VLIW. These types of optimizations were explored later in [10, 21]. VLIW architectures present an opportunity for testing the effectiveness of compiler optimizations. Unlike superscalar processors, VLIW processors rely on compiler for effective instruction scheduling.
Chapter 9

Conclusion

9.1 Data Dependence Profiling

Data dependence profiling refers to the collection of data dependence information at runtime. This information provides a better understanding which can be utilized to unlock performance potential within the code by releasing conservative constraints. In this thesis, we describe the challenges in building a data dependence profiler, implement and evaluate prior attempts at building such a profiler, and propose a novel set-based approach to data dependence profiling which significantly outperforms prior approaches in speed and accuracy. Subsequently, the utility of this data dependence information is tested in a feedback directed optimization framework built using a state-of-the-art compiler.

Data dependence profiling is challenging since it requires determining the relationships between loads and stores in the program. Since about 30-40% of all instructions are loads or stores, and without static analysis, any two memory operations can alias with one another, this problem quickly becomes very tough to solve, especially when it comes to large programs.

Prior approaches did not utilize static analyses, and hence worked under the assumption that any two instructions in the program can potentially alias. This led to data dependence profilers which utilized centralized storage to track all dependences in the program(region). This led to slow and/or inaccurate profiles. Prior works have shown that not all dependences are relevant. And static analysis can determine the nature of relationships between any two memory operations. Only those operations that are both relevant and could alias, need to be checked. This approach leads towards a decentralized storage solution. In order to achieve, and beat the performance of the centralized storage solution, the allocation of this decentralized storage needs to be smart.

In this vein, we propose a novel set-based data dependence profiler that significantly reduces the cost of collecting accurate data dependence profiles. The set profiler utilizes static analysis
to gather information regarding the queries, and uses smart set allocation schemes to accurately profile the queries provided. The sets provide decentralized storage, which is managed through set operations inserted by the compiler. The profile runs are quick, and the information is accurate.

An initial version of the profiler was able to profile programs from the SPEC benchmark suite with a slowdown of 8x, with good accuracy (0.15NAED). This version of the profiler relied on a hill-climb based design space exploration to achieve this performance and accuracy point. Our evaluations in this phase showed that the profiler could traverse a large space of performance and accuracy tradeoffs. More importantly, the profiler’s behavior was predictable once the programs, and the set profiler’s knobs were understood. This knowledge allowed us to further optimize the compiler with novel optimizations.

*Early Termination* This optimization allows the profile to detect redundant checks on-the-fly and disable them, hence decreasing overhead without affecting accuracy. We found that this technique helps traditional profilers improve their performance without a loss of accuracy. This technique was able to speed up the profiler by a factor of 2, against the BestSig configuration with 8x slowdown.

*Edge Profile Set Sizing* This optimization tuned each individual set used by the profiler, according to its population. This tuning allowed us to use different types of signatures for different sets within the same profile run. This optimization, when coupled with Early Termination, sped up the profiler by 4x, against the BestSig configuration, however there was a significant deterioration in accuracy.

*AllocS Set Allocation* The basic and Root Sets set allocation schemes assign sets based on the interferences in the dependence graph. In order to make set allocation schemes aware of the population counts within, we introduced the AllocS set allocation. In addition to the Root Sets allocation, this scheme limited the number of instructions entering a single set. This allowed the sets to maintain accuracy even when their sizes were decreased. Coupled with the Early Termination, this configuration improved performance when compared to BestSig, by a factor of 4, with a negligible loss in accuracy.

### 9.2 Feedback Directed Optimization

In order to demonstrate that there exists untapped performance in programs, and that it can be realized through profile information, we built a feedback-directed optimization framework using a state-of-the-art compiler. The framework is a standalone tool, which is compatible with LLVM-3.1. The framework includes the following profilers.

- A global edge profile
• A data dependence profiler which can profile
  – No-aliases using the set-based profiler
  – Must-aliases using a simple value-checking mechanism

In addition to these profilers, the framework also includes a few passes which directly utilize the profile information to transform the code. These passes represent examples for further speculative optimizations which can harness the full potential of the profile information provided.

Since we are trying to explore the potential for optimization, the tool assumes zero cost recovery. In our tests, we use the same input for profiling as well as execution. This obviates the need for a recovery mechanism.

Our results show that there is significant performance that can be had by unlocking conservative constraints in programs. At a high-level, the performance numbers represent the impact of writing code which can handle multiple inputs. This over-provisioning of the code tends to hinder compilers from performing optimizations, in order to maintain semantic correctness under all inputs. We relax such constraints by restricting the input set to the profiling input.

Upon feedback, edge profile tends to decrease, and simplify, the control-flow within the program, thus leading to better code generation. Data dependence profiles tend to highlight opportunities for finding redundancies within the code. Both the profiles are necessary in certain benchmark in order to unlock performance.

This thesis proposed a novel approach to data dependence profiling that is faster and more accurate than any prior art. This profiler was incorporated in a feedback-directed optimization framework, along with other profilers to demonstrate the effectiveness of profile information in speculatively optimizing programs.
References


Appendix
Appendix A

Tool Description

This chapter describes the tool that implements the profiler and the speculative optimizations evaluated in this thesis. This is a standalone tool that was built using the hooks provided by LLVM 3.1, and works on the IR-level bytecode. There are two main parts to the tool, namely, the profiler and the speculative optimization framework. We’ll first describe the profiler, and subsequently, the speculative framework.
A.1 Profiler

Figure A.1: Code Layout for Profiling

Figure A.1 shows the code layout for the profiler part of the tool. This figure shows the various stages leading up to the instrumentation of the bytecode.

Each of the stages is implemented as a pass/interface within the tool. The big boxes indicate the interface for that stage, whereas the smaller rectangles below each box indicate the implementations of this interface. In the case of the Query Interface, both the Global Code Motion Queries and Must Alias Queries provide queries. Both these passes can be enabled simultaneously to analyze both types of queries in the same profile run. In the other stages, only one implementation is enabled at any given time.

For all the experiments shown in the thesis, the Root Sets Interference Graph was used. We evaluated multiple types of profilers using the different implementations of the same instrumentation interface. Perfect Instrument refers to the Ideal Profiler. Perfect Set Instrument provides the same accuracy as the Ideal Profiler, but unlike the Perfect Profiler, this uses sets.
A.2 Feedback Directed Optimizations

Figure A.2 shows the code layout for the feedback part of the tool. This figure shows the various stages making up the feedback part of the framework. The QueryInterface is used to gather all the queries that were profiled. Profile data for these queries is provided, along with the queries themselves, to the Feedback pass. This pass maps the profile data to the queries, and uses the Alias Analysis interface to expose this information to the optimizations.

Now, the queries can be filtered in the Feedback, based on heuristics to decide if they are useful to optimizations. We use a simple heuristic, a threshold for the alias probability, to find relevant queries. Leveraging the Alias Analysis in LLVM allows us to provide this profile information to a multitude of optimization passes with little or no modifications. The optimizations perform speculative transformations on the code, without any explicit changes to them, and the output is then passed to the backend. Currently, the profile information is not visible to the backend, as the mapping is lost once the code is lowered.