ABSTRACT

HUH, JOONMOO. Using Runtime Value-range Invariants to Optimize The Bit Width of Data for Memory Usage Efficiency. (Under the direction of James Tuck.)

As software systems become larger and more complex, inefficient use of data increases due to the relative abundance of system memory. Such inefficiency degrades the performance of the system by increasing memory traffic and under-utilizing the cache hierarchy. This thesis proposes a tool that dynamically detects value-based invariants by tracking the ranges of values held in source-level variables. By detecting the actual range of values for a variable, a programmer can change the storage requirements of a variable by altering its type declaration. I identify a few simple rules to replace variables of a larger type with ones of a smaller type, that if followed, make it easy for the programmer to optimize the code.

Using the tool, I analyzed the Mibench and SPEC CPU2006 benchmarks, and found that the fraction of wasted bytes is surprisingly high. The analysis result showed that 77% of 4-byte typed variables out of all tracked 4-byte typed variables in the benchmarks can be replaced to smaller data types. After optimization, 10.1% fewer bytes are accessed in memory, on average, than the original program. I also simulated the behavior of a cache for 5 sets of the benchmarks which miss more than 0.1% in L1 cache. For those applications, my strategy reduces the miss rate of the L1 data cache by 17.5% on average.

© Copyright 2012 by Joonmoo Huh

All Rights Reserved

Using Runtime Value-range Invariants to Optimize The Bit Width of Data for Memory Usage Efficiency

by
Joonmoo Huh

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Electrical Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Alexander Dean	Huiyang Zhou

James Tuck
Chair of Advisory Committee

DEDICATION

To my parents, Mr. Kyuhyeng Huh and Mrs. Sooneui Koo
and my wife, Hyunkyoung Lee
for their support and love.

BIOGRAPHY

Joonmoo Huh was born in Seoul, South Korea on the 11th of December 1982. He obtained his B.S. in Electronic Engineering from Inha University in Incheon, South Korea in 2008. He worked as a researcher at the Intelligent Embedded System Research Laboratory, Inha University, where he studied about improving reliability in a mobile storage system. He joined North Carolina State University in 2010 to pursue his M.S. degree. During his M.S. study, he has focused his research on using runtime value-range invariants to optimize the bit width of data for memory usage efficiency under the direction of Dr. James Tuck. On June 14, 2012, he defended his M.S. Thesis.

ACKNOWLEDGEMENTS

First, I would like to thank Professor James Tuck, my advisor, for his advice and feedback. Professor Tuck always gave insightful comments about my work. I also would like to thank my Ph.D committee members, Dr. Alexander Dean and Dr. Huiyang Zhou for their suggestions and encouragement.

I would like to thank all the CESR student colleagues. Specially, Sangyeol Kang shared research ideas and personal life with me. In addition, I would like to thank Liang Han, Sanghoon Lee, George Patsilaras, Rajeshwar Vanka and Ying Yu for discussion about research ideas. I would like to thank all the Korean friends I have met in Raleigh since 2010. I have been so happy enjoying my life with them.

Last but not least, I would like to express my sincere appreciation and gratitude to my father, mother and wife for the tremendous support. I'll never forget their love and sacrifice.

TABLE OF CONTENTS

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Chapter 1 Introduction **1**
1.1 Overview 1
1.2 Motivation 1
1.3 Related works 3
1.3.1 Invariant-rule-based runtime monitoring 3
1.3.2 Value profiling 3
1.3.3 Bit-width Minimization 4
1.4 Outline 4

Chapter 2 Detecting Value-range Invariant **5**
2.1 Source-level instrumentation 5
2.2 Data structures 5
2.2.1 Meta-data 5
2.2.2 Hash table 6
2.3 Updating the Meta-data 7
2.4 Instrumented points 8

Chapter 3 The Best-fit Type **9**
3.1 Definition 9
3.2 Optimization methodology 9
3.2.1 Semantic issue 10
3.2.2 Input sets issue 10
3.2.3 Performance issue 10

Chapter 4 Implementation **11**
4.1 Source-to-source translator 12
4.2 Instrumented points 13
4.3 Dynamic analysis 13
4.4 Coverage 14

Chapter 5 Evaluation **16**
5.1 Benchmarks analysis 16
5.1.1 MiBench 16
5.1.2 SPEC CPU2006 16
5.1.3 Analysis 16
5.2 Result 19
5.2.1 Estimated coverage 19
5.2.2 Variable types composition 19
5.2.3 Memory accesses composition 20

5.2.4 Total bytes of memory accesses 20
5.2.5 Cache miss rate 23

Chapter 6 Conclusion **24**
6.1 Conclusion 24
6.2 Future work 24

References **25**

LIST OF TABLES

Table 3.1 Size of integer types in C language on the target system 9

Table 4.1 Examples of instrumentation rules 14
Table 4.2 Examples of instrumentation 15

Table 5.1 MiBench benchmark suites and input sets 17
Table 5.2 SPEC CPU2006 benchmark suites and input sets 17
Table 5.3 Analysis of benchmarks 18

LIST OF FIGURES

Figure 2.1 Example of the meta-data 6
Figure 2.2 Example of the hash table 7
Figure 2.3 Example of updating the meta-data 7

Figure 4.1 Overview of the implementation 11
Figure 4.2 Clang driver phases 12
Figure 4.3 Bound Clang driver phases for the source-to-source translator 12

Figure 5.1 Estimated coverage in memory accesses 19
Figure 5.2 Change of variable type composition at the source-level 20
Figure 5.3 Change of memory access composition without optimization option 21
Figure 5.4 Change of memory access composition with -O3 optimization option . . . 21
Figure 5.5 Total bytes of memory accesses 22
Figure 5.6 L1 cache miss rate 23

Chapter 1

Introduction

1.1 Overview

As software systems become larger and more complex, inefficient use of data will likely increase. The inefficiency results not only from complicated data structures in the system, but also from the relative abundance of system memory. For example, assigning a type with greater than necessary is a kind of inefficient memory usage. Programmers often pay insufficient attention to this small detail because (1) it is not always simple to find the most efficient type for each variable in a program, and (2) they can use relatively much larger system memory today than ever before. Furthermore, it is very tedious work if the program handles a lot of variables. However, inefficient type usage leads to performance degradation by increasing memory traffic and under-utilizing the cache hierarchy.

In this thesis, I introduce a tool that detects the range of values held in all variables in a program using source-level instrumentation. Using these range-invariants, the most efficient type for the variables tracked can be identified. I analyzed benchmark suites from MiBench and SPEC CPU2006, and showed that the fraction of wasteful declarations, variable declarations using too large of a storage type, is surprisingly high. By replacing wasteful declarations with the minimal declaration required by the invariant, I reduced the total bytes accessed in memory and the miss rate of the L1 data cache.

1.2 Motivation

Programming languages provide a variety of types with different ranges and storage sizes. It is obvious that a type of a variable should be carefully chosen considering the range of a value the variable has over the lifetime of each variable to achieve better performance with minimum storage. However, when it comes to assigning a type to a variable, it is very dangerous to assign

a small type which is not large enough to handle the data because the correctness of the program is not guaranteed. On the other hand, assigning a bigger type to a variable has no effect on the correctness. Therefore, programmers tend to choose a type which is too large to store the data over the lifetime of the variable when the minimum size of the type for the variable cannot be easily calculated.

The following C code below gives the example of a variable assigned a bigger type. The code is from *bzip2* in SPEC2006 benchmarks [11]. In the source code, *runningOrder* at line 3 is declared as 32-bits signed integer type which can represent the number from -2,147,483,648 to 2,147,483,647. However, the value of *runningOrder* never exceed the range from 0 to 255 unless the value is changed by *BIGFREQ* macro at line 13.

```
1  static void mainSort (UInt32* ptr, UChar* block, ... ) {
2     Int32 i, j, k, ss, sb;
3     Int32 runningOrder[256];
4     ...
5     for (i=0; i<=255; i++) {
6        bigDone[i] = False;
7        runningOrder[i] = i;
8     }
9     ...
10    for (i=h; i<=255; i++) {
11       vv = runningOrder[i];
12       j = i;
13       while (BIGFREQ(runningOrder[j-h])>BIGFREQ(vv)) {
14          runningOrder[j] = runningOrder[j-h];
15          j = j - h;
16          if (j<=(h-1)) goto zero;
17       }
18       zero:
19       runningOrder[j] = vv;
20    }
21    ...
```

In fact, the type of *runningOrder* can be replaced with *unsigned char* type because the *BIGFREQ* macro does not change the value of its argument. If the programmer assigned *unsigned char* type to *runningOrder*, the program could save 3 bytes (4 bytes - 1 byte). In addition, *runningOrder* is a 256-entries array type, so the variable not only uses 768 bytes less memory, but also has a better locality since all data of the variable is located in a smaller area in memory.

1.3 Related works

1.3.1 Invariant-rule-based runtime monitoring

An invariant is a rule that a program obeys at a particular program point or points. There has been some research in using run-time invariants. AccMon [20] detects bugs using program counter (PC)-based invariants. DIDUCE [10] detects bugs and identifies the root causes of the bugs by checking program invariants dynamically. AVIO [13] detects atomicity violation bugs using access interleaving (AI)-based invariants. Swarup K. Sahoo [16] also uses value-based invariants to detect hardware errors. All of theses research only focus on detecting bugs or errors while I focus on giving optimization opportunities to programmers.

Invariant detection techniques have been studied by Michael D. Ernst et al. [6]. The techniques are implemented in the Daikon tool [7] which provides 75 different invariants for programs written in C, C++, Java and Perl. The tool runs with inputs which are traces of the data generated from a program. The program should be compiled with DWARF-2 format debugging information if the program is written by C language. Because of the large overhead required for storing and handling the traces and the debugging information, invariants are only detected at procedure entries and exits, and the coverage of variables is constrained to procedure parameters, return values and global variables. On the other hand, my tool focuses on a specific invariant, the range of values for a variable using source-level instrumentation. The tool instruments original source code to be able to derive the invariants without additional traces and debugging information.

1.3.2 Value profiling

B. Calder et al. dynamically identify semi-invariant variables to give benefit to several areas of value-based compiler optimization [4]. They categorize invariants according to instruction types, store top N value of the variables and find the semi-invariant variables in the instructions. They explore the invariant behavior from profiling of load instructions and memory locations [5]. G. Xu et al. [18] introduce a dynamic analysis to find inefficient data structures by profiling the numbers of nodes and edges in terms of space overhead. In fact, they focus on profiling data structures rather than optimization schemes. Several value-based compiler optimizations, such as constant propagation, code specialization and partial evaluation, have been studied using value profiling [8, 12]. However, none of them is interested in the wasteful type declarations at the source-level.

1.3.3 Bit-width Minimization

There has been interest in analyzing the bit-width of data to help Silicon compilers or to support hardware accelerators [17, 15]. They rely on static analysis at the assembly-level or intermediate representation, while our dynamic analysis is based on source-level instrumentation. Brooks and Martonosi [3] improved performance by dynamically merging narrow width operands, and Zhang and Gupta [19] reduced heap allocated storage by designing data compression extensions (DCX). However, these schemes need hardware support while my scheme can be completed without any modifications to a conventional compiler or a hardware since I optimize source code.

1.4 Outline

The thesis is organized as follows. Chapter 2 provides the information on the value-range invariants. In addition, I also shows the data structures I designed for deriving the invariants and detecting algorithm using source-level instrumentation. Chapter 3 gives the methodology of optimization considering the invariants. Chapters 4 details the implementation of detecting invariants and the framework for dynamic analysis. Chapter 5 shows experimental results of my tool and optimization strategy. I close with future work in Chapter 6.

Chapter 2

Detecting Value-range Invariant

This chapter describes a dynamic value-range invariant detection algorithm using source-level instrumentation.

2.1 Source-level instrumentation

Since the purpose of deriving invariants in this research is to give programmers feedback on the variables which are assigned to a wasteful type, it is important to find the minimal and maximal value ever stored in a variable. This is accomplished by tracking all declared and allocated variables while a program is running; each time the variable is updated, their minimum and maximum values are also updated as meta-data.

A source-to-source complier is developed to profile the range of values. The complier instruments the original source code with the profiling algorithm. Thus, the instrumented source code can be compiled to the instrumented executable binary, and the value-range invariants are derived without any traces or debugging information by just running the instrumented program.

2.2 Data structures

The program is instrumented with two data structures to derive the range invariants: Meta-data and Hash table.

2.2.1 Meta-data

The source-to-source complier pass adds code to track meta-data for each variable declaration while the instrumented program runs. Each meta-data entry is identified with a unique serial number, the type information for the variable and two additional storage locations for the maximum and minimum value ever observed. Note, a serial number is unique for each symbol

in the original source code. The meta-data is created at the starting point of a program through static analysis. An example of the meta-data is shown in Figure 2.1. Assuming no other variables are analysed first, the meta-data will be filled out like the entries right-side in the figure.

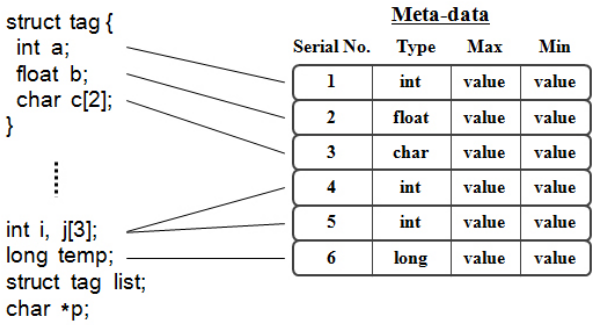


Figure 2.1: Example of the meta-data

The algorithm is not limited to tracking scalar variable declarations in the meta-data. *Struct* types are also handled by creating the meta-data for each field of the *struct* declaration. All instances of the *struct* share a meta-data entry. This is overly conservative but keeps our methodology simple for now.

2.2.2 Hash table

The other data structure is a hash table which maps the address of a variable declaration to its serial number in the meta-data. Each hash table entry stores the address of a variable declaration as *key* and the serial number of the declaration as *value*. The hash table is dynamically updated according to the lifetime of the variable declarations. The hash table is important because it allows us to map accesses through pointers back to their original source level declaration. The other reason why the hash table is important is that multiple copies of a source-level variable could exist at a given program point and all of them should update the same meta-data. For example, a recursive function while in recursion would have multiple instances of the same source level variable. Similarly, if there were an array of type struct, all elements in the array would update the same meta-data for their particular field. The hash table maps all such addresses back to a common meta-data entry. An example of the hash table is shown in Figure 2.2. Assuming the no other variable declarations are stored first, the hash table will be filled out like the entries right-side in the figure.

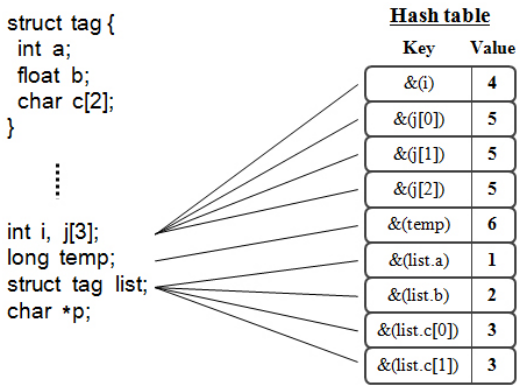


Figure 2.2: Example of the hash table

2.3 Updating the Meta-data

The meta-data is updated whenever its corresponding variable is changed. First, the hash table is indexed using the address of the accessed variable. Then, the hash table points back to the serial number of the meta-data. Finally, the max and min range of the variable declaration is dynamically updated. Figure 2.3 shows how the meta-data is updated when a value of variable is assigned.

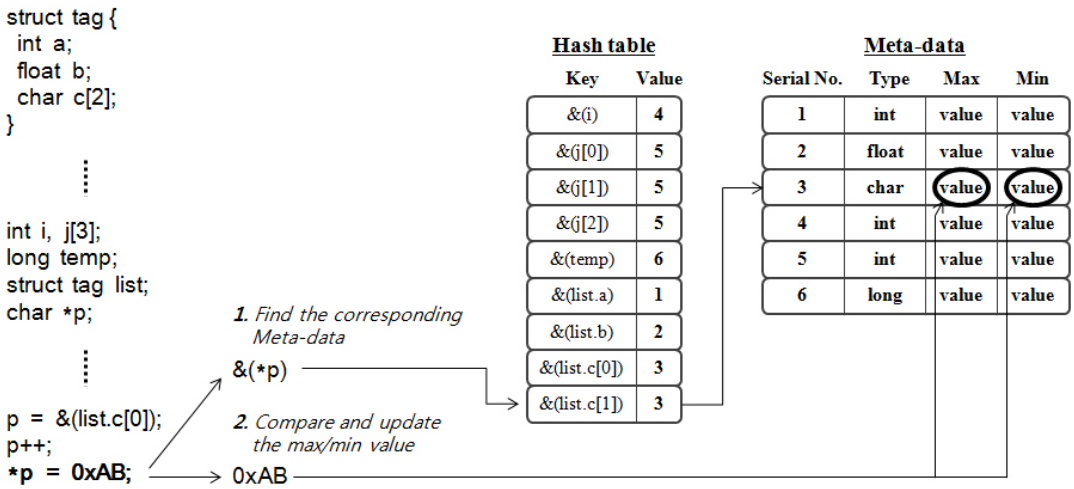


Figure 2.3: Example of updating the meta-data

2.4 Instrumented points

The instrumented program has seven kinds of points where the source code is inserted or rewritten to create and update the meta-data. The points are described as follows.

1. **At the starting point of a program:** The meta-data for all targeted variables is initialized.
2. **At the starting point of functions:** The information for each variable declarations in the first declaration group is added to the hash table.
3. **At the exit points of functions:** The information for all variable declarations in the dominating declaration groups in the function are deleted from the hash table.
4. **At an allocation of memory dynamically:** The information for the allocated variable is added to the hash table.
5. **At a deallocation of memory:** The information for a freed variable is deleted from the hash table.
6. **At a variable update:** The meta-data for the variable is updated by looking up its address in the hash table and using the serial number found there to update the max and/or min value.
7. **At the exit points of a program:** All invariants in the meta-data are stored to file.

Chapter 3

The Best-fit Type

In this section, the optimization scheme to assign the best-fit type for a variable is described.

3.1 Definition

There are four basic arithmetic types (*char*, *int*, *float*, *double*) with optional specifiers (*signed*, *unsigned*, *short*, *long*) in C language. For each variable used in a program, *the best fit type* is defined as the minimum-sized type which can store all range of number that the variable uses. Table 3.1 shows that the size of integer types in a 32-bit Linux machine when a program compiled by gcc 4.4.3.

3.2 Optimization methodology

Considering the range of a variable, the original declared type can be replaced to the best fitting type. However, The replacement have to be carefully done by looking at the source code

Table 3.1: Size of integer types in C language on the target system

Type	**Size**	**Range(unsigned)**	**Range(signed)**
char	8 bits	$0 \sim 255$	-128 $\sim$ 127
short int	16 bits	$0 \sim 65{,}535$	-32,768 $\sim$ 32,767
int, long int, float	32 bits	$0 \sim 4{,}294{,}967{,}295$	-2,147,483,648 $\sim$ 2,147,483,647
long long int, double	64 bits	$0 \sim (2^{64} - 1)$	$-2^{63} \sim (2^{63} - 1)$

when replacing types. In the experiments of this thesis, all wasteful declarations are manually replaced with their best fitting types considering the three issues mentioned below.

3.2.1 Semantic issue

Although the range of a variable fits into a smaller type than the original declared type, it is not safe to replace the current type with the best fitting type for semantic reasons. For example, if a variable's address is taken, a pointer may be used to access it. If the pointer also refers to other variables whose types cannot be changed to match the same best fitting type, then the pointer will have inconsistently typed targets. Clearly, this incorrect and should be avoided.

3.2.2 Input sets issue

The replacement of types should be done with a deep understanding of the source code since the dynamic analysis does not guarantee the invariants will hold for other inputs which have not been analyzed. However, the replacement will be absolutely safe if the program runs with only input sets which have been analyzed. In this thesis, this issue is ignored by assuming that all possible input sets are analyzed.

3.2.3 Performance issue

The replacement of types could hurt the performance of the program. Even though changing the type reduces the memory traffic and occupancy of memory, it may cause performance degradation by adding extra type casting operations. In this thesis, this issue is handled by focusing on reducing the number of memory accesses. The replacements which cause the increase of memory accesses are skipped.

Chapter 4

Implementation

The overview of the implementation is shown in Figure 4.1. The first row presents the instrumentation to detect invariants. Two instrumentation phases are used because all macros should be first eliminated before instrumenting the code to detect invariants. In the first phase, all preprocessor macros in all of the source files of the program are replaced. The invariant detection scheme is instrumented in the second instrument phase. These two phases are implemented by the source-to-source translator I developed. Finally, the instrumented program can dynamically find and manage invariants.

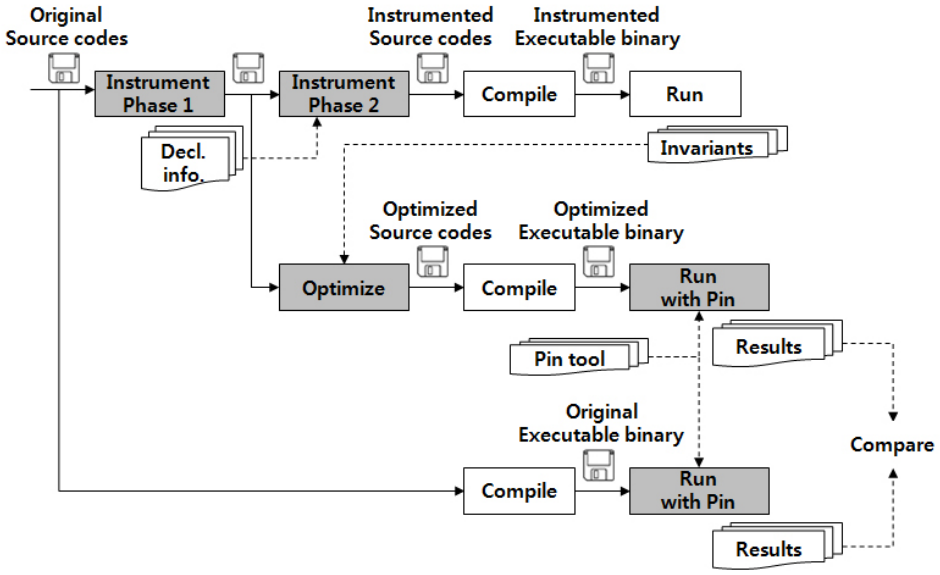


Figure 4.1: Overview of the implementation

The manual optimization phase is shown on the second row. All invariants are merged across all input sets for matching serial numbers (same source-level variable). Using the merged invariants, the type of a variable is replaced to the best fitting type, if the original type is different from the best fitting type. After manually changing the code, the new optimized source code is compiled into the final optimized binary.

The third row shows the compilation of the original program. To compare characteristics of the original program with the optimized program, Pin tool [14] is used for characterizing the number and size of accesses to memory across the entire program's execution.

4.1 Source-to-source translator

I developed a source-to-source instrumentation using Clang [1]. Clang has a production quality compiler driver providing access to its compiler and tools, with a command line interface which is compatible with the *gcc* driver. Figure 4.2 shows the phases of the Clang driver when a simple source file is complied. The dark circles represent the phases in the Clang driver, and the bright circles are the inputs/outputs of each phase. The Clang executable binary file covers all of these phases except the linker phase. All of these phases in the Clang project were designed to be flexible and easily accommodate new uses.

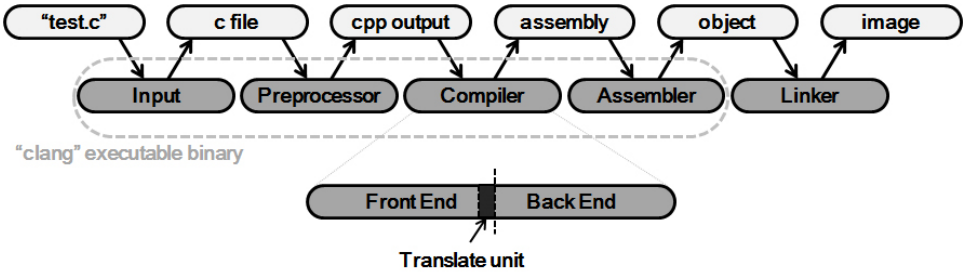


Figure 4.2: Clang driver phases

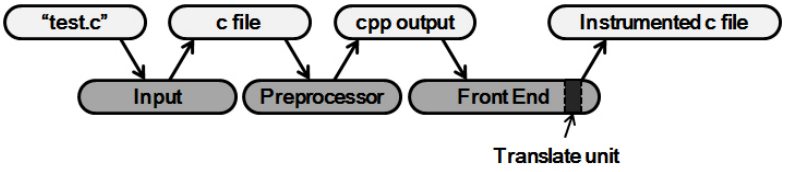


Figure 4.3: Bound Clang driver phases for the source-to-source translator

A source-to-source compiler for this research was implemented by binding a portion of the overall sequence from the input phase to the front-end of the compiler phase in the Clang project. This is shown in Figure 4.3.

4.2 Instrumented points

There are seven kinds of points where the source code is inserted or rewritten as mentioned before. The locations of the points at the source-level are described as follows.

1. **Starting point of a program:** A meta-function is inserted at the end of the first declaration group in the *main* function.
2. **Starting point of functions:** A meta-function is inserted at the end of the first declaration group in the functions.
3. **Exit points of functions:** A meta-function is inserted at the end of the functions. Note, there may be several points to exit a function and all are instrumented.
4. **Allocation points of memory dynamically:** There may be several kinds of dynamic allocation; I only track calls to *malloc* and *calloc*, but custom allocators could be supported easily. *Call* expressions for memory allocation function are instrumented as shown in Table 4.1.
5. **Deallocation point of memory:** A meta-function is inserted at the *call* expressions for *free* function. *Call* expressions for *free* function are instrumented as shown in Table 4.1.
6. **Variable update points:** There are three kinds of points to write accesses at the source-level: *assignment* operators and *increment/decrement unary* operators. Variables may also be modified due to pass-by-reference in C, but that will eventually be caught in an assignment. *assignment* operators and *increment/decrement unary* operators are instrumented as shown in Table 4.1
7. **Exit point of a program:** A meta-function is inserted at the end of *main* function and the *call* expressions for *exit* function

Various examples of instrumentation above are shown in Table 4.2.

4.3 Dynamic analysis

To analyze the optimizations, the number of memory accesses is measured and categorized by the size in bytes of their accesses. A cache analysis is also performed by integrating Dinero IV

Table 4.1: Examples of instrumentation rules

Original statement	**Instrumented statement**
malloc($expr$)	meta_alloc(malloc($expr$), $expr$)
calloc($expr_1$, $expr_2$)	meta_alloc(calloc($expr_1$, $expr_2$), $expr1 * expr_2$)
free($expr$)	(free($expr$), meta_free($expr$))
$expr_1 = expr_2$	assign_XX(&($expr_1$), $expr_2$)
$expr_1 += expr_2$	assign_XX(&($expr_1$), $expr_1 + expr_2$)
$expr_1 -= expr_2$	assign_XX(&($expr_1$), $expr_1 - expr_2$)
$expr++$	unary_inc(&($expr$), $true$)
$expr--$	unary_dec(&($expr$), $true$)
$++expr$	unary_inc(&($expr$), $false$)
$--expr$	unary_dec(&($expr$), $false$)

[2] to the Pintool. The core function of Dinero IV is directly integrated into the pintool so both the result of the dynamic analysis and the cache simulation are obtained in one run.

4.4 Coverage

The coverage of the tool in this research is explained by two categorizes: type and scope. All basic type variables in C language are tracked in terms of types. In addition, it can handle arrays up to two dimensions and member variables of structures including nested structure. The types that are not tracked by this tool are pointer, enumerated and unioned type. Second, the tool has the ability to track the following kinds of variables:

1. Global variables: Variables which are declared outside of functions in the source file which has *main* function.

2. Local variables: Variables which are declared at the first declaration group in each function.

3. Dynamic variables: Variables which are allocated and freed using *malloc*, *calloc* and *free* function calls.

The write accesses which occur in a library function are not tracked by the tool. Since untracked calls, such as *memset*, *memmove* or *memcpy* function, could hurt the accuracy of the final invariant, some library functions which are known to write to memory are detected and instrumented at their call site.

Table 4.2: Examples of instrumentation

Original source code	Instrumented source code

`int main (int argc, char argv[]) {`
`  char c = 1;`
 	
`  while (c) {`
`  ...`
`  exit(1);`
`  ...`
`  return 0;`
`}`
	`int main (int argc, char argv[]) {`
	`  char c = 1;`
	
	`  meta_program_start();`
	`  meta_main_start(&c);`
	`  while (c) {`
	`  ...`
	`  (meta_program_end(), exit(1));`
	`  ...`
	`  return (meta_main_end(&c), \`
	`  meta_program_end(), 0);`
	`}`

`void foo (int a, int b) {`
`  int tmp[32];`
 	
`  tmp[15] = a;`
`  ...`
`  if (tmp[31]) return;`
`  ...`
`}`
	`void foo (int a, int b) {`
	`  int tmp[32];`
	
	`  meta_foo_start(&a, &b, tmp);`
	`  assign_int(&(tmp[15]), a);`
	`  ...`
	`  if (tmp[31]) {`
	`  meta_foo_end(&a, &b, tmp);`
	`  return;`
	`  }`
	`  ...`
	`  meta_foo_end(&a, &b, tmp);`
	`}`

`  int i;`
`  ...`
`  for (i=0; i<256; i++) {`
`  ...`
	`  int i;`
	`  ...`
	`  for (assign_int(&(i), 0); i<256; \`
	`  unary_inc(&(i), 1) {`
	`  ...`

`  char a, b, c;`
`  ...`
`  c = fee();`
`  a = b = c;`
`  ...`
	`  char a, b, c;`
	`  ...`
	`  assign_char(&(c), fee());`
	`  assign_char(&(a), assign_char(&(b), c));`
	`  ...`

Chapter 5

Evaluation

For this research, The evaluation was performed on a Linux virtual machine system with an Intel i5 processor clocked at 2.53GHz with 4GB memory running 32-bit Linux 2.6. Benchmarks were compiled by gcc 4.4.3 both with no optimization and with -O3 optimization. For the cache simulator, I modeled a 2-way 32KB set-associative cache with a 64 byte block size.

5.1 Benchmarks analysis

5.1.1 MiBench

MiBench [9] is a set of commercially representative embedded benchmarks written in the C language. The benchmark suites in MiBench are categorized by six sections: Automotive and Industrial Control, Consumer Devices, Office Automation, Network, Security and Telecommunications. Two or three benchmarks are chosen for evaluation from each section. Table 5.1 shows the summary of the benchmark suites and input sets.

5.1.2 SPEC CPU2006

SPEC CPU2006 [11] is the benchmarks that aim to test *real life* situations and to test CPU performance by measuring the time of several programs. Four benchmarks are chosen for evaluation. Table 5.2 shows the summary of the benchmark suites and input sets.

5.1.3 Analysis

The tool is able to detect invariants for all variables which are actually written while the program runs. The number of invariants is highly dependent on the characteristics of program. For example, although the program has a lot of variable declarations, the number of invariants could be small if only a few variables are used over the lifetime of the program.

Table 5.1: MiBench benchmark suites and input sets

Auto.	**Consumer**	**Office**	**Network**	**Security**	**Telecomm.**
basicmath	**jpeg**	**ispell**	**dijkstra**	**rijndael**	**FFT**
-default	*-encode*	*-default*	*-default*	*-encode*	*-forward*
bitcount	*-decode*	**rsynth**	**patricia**	*-decode*	*-inverse*
-default	**tiff**	*-default*	*-default*	**sha**	**adpcm**
susan	*-tiff2bw*	**string**		*-default*	*-encode*
-edges	*-tiff2rgba*	**search**			*-decode*
-corners	*-dither*	*-default*			**gsm**
-smooth	*-median*				*-encode*
					-decode

Table 5.2: SPEC CPU2006 benchmark suites and input sets

400.perlbench	**401.bzip2**	**403.gcc**		**429.mcf**
-test	*-liberty.compress*	*-166*	*-expr2*	*-test*
	-liberty.uncompress	*-200*	*-g23*	*-train*
	-chicken.compress	*-cccp*	*-integrate*	
	-chicken.uncompress	*-cp-decl*	*-s04*	
	-combined.compress	*-c-type*	*-scilab*	
	-combined.uncompress	*-expr*		

Table 5.3: Analysis of benchmarks

Benchmark	**Invariants**	**Wasteful decls**	**Opt. decls**	**Overhead**
basicmath	15	2 (13.3%)	1 (6.7%)	4.9x
bitcount	21	8 (38.1%)	8 (38.1%)	112.0x
susan	76	63 (82.9%)	59 (77.6%)	58.8x
jpeg	403	365 (90.6%)	344 (85.4%)	54.1x
tiff	216	182 (84.3%)	159 (73.6%)	29.9x
ispell	56	37 (66.1%)	35 (62.5%)	1103.1x
rsynth	93	66 (71.0%)	66 (71.0%)	91.9x
stringsearch	4	3 (75.0%)	3 (75.0%)	66.4x
dijstra	15	15 (100%)	15 (100%)	33.7x
paticia	6	3 (50.0%)	3 (50.0%)	7.0x
rijndael	19	11 (57.9%)	11 (57.9%)	12.8x
sha	21	10 (47.6%)	10 (47.6%)	121.5x
FFT	19	15 (78.9%)	15 (78.9%)	9.9x
adpcm	27	19 (70.4%)	18 (66.7%)	117.4x
gsm	113	59 (52.2%)	41 (36.3%)	18.5x
perlbench	358	302 (84.4%)	241 (67.3%)	30.6x
bzip2	196	94 (48.0%)	93 (47.4%)	62.8x
gcc	1952	1596 (81.8%)	1512 (77.5%)	-[1]
mcf	64	35 (54.7%)	33 (51.6%)	252.4x
Average	193.8	151.8 (65.6%)	140.4 (61.6%)	47.2x

The number of invariants the tool derives is shown in Table 5.3. By considering the invariants, it is discovered that 65.6% of tracked variables are assigned a wasteful type declaration on average. I optimized 61.5% of tracked variable on average by replacing its wasteful type declaration to best-fit type declaration. The overhead of our instrumentation is also measured by counting the number of instructions executed. The overheads of each benchmark and their geometric average are also shown in Table 5.3. The overhead of *ispell* benchmark from *Office automation* category in MiBench is incredibly high, 1103.1x. This is because the benchmark has several tremendous arrays which are accessed a lot and the hash table performs poorly.

[1] I skipped to measure the exact overhead of gcc benchmark due to the time limit. The overhead was more than 1500.0x

5.2 Result

The optimization scheme is evaluated only on Mibench benchmark suites. It is very hard to apply manually the optimization scheme to the benchmark suites from SPEC CPU2006 because they have relatively large number of invariants and more complicated program structure.

5.2.1 Estimated coverage

As the instrumentation in this research focuses on tracking the write access points, the coverage of this tool can be estimated by counting tracked write accesses and total write accesses. Figure 5.1 shows the estimated coverage in memory accesses. The number of tracked write accesses is normalized by the total number of write accesses over the lifetime of the programs. Average estimated coverage is 36.55%. Naturally, the coverage is highly dependent on the characteristic of program. For example, the accesses of pointer variables are excluded from the coverage because the tool focuses on the value of arithmetic variables rather than pointer variables.

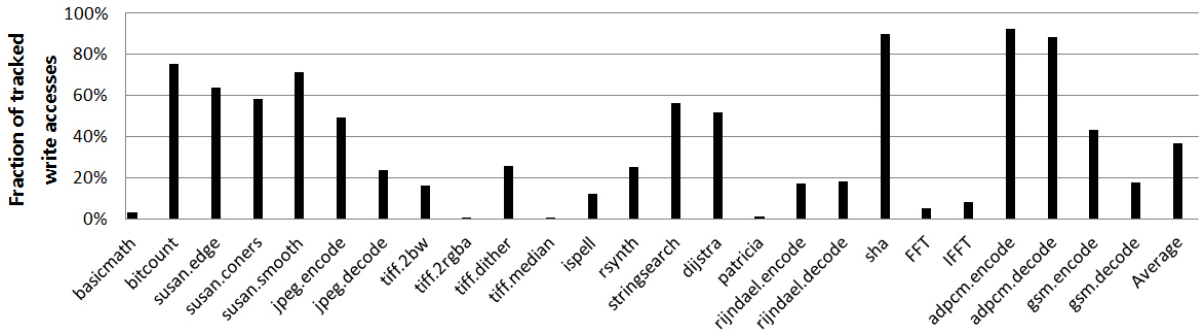


Figure 5.1: Estimated coverage in memory accesses

5.2.2 Variable types composition

The change of variable type composition between original programs and optimized programs is shown in Figure 5.2. For each benchmark, out of the total number of traced variables, the left bar shows the percentage composition of variable types in the original program while the right bar indicates the percentage composition of variable types in the optimized program. 4 byte types (*int*, *long*, *float*) are used most frequently in the original program on average. However, only 23% of the 4 byte types are used as the best fitting type. On the other hand, the percentage composition of *char* and *short* types are only 8.48% and 6.23% respectively in the original program, but the optimized programs assign 52.84% of *char* types and 23.54% of

short types respectively. This shows that a lot of variables in programs use unnecessary space because of careless type assignment. In aggregate, this waste adds up.

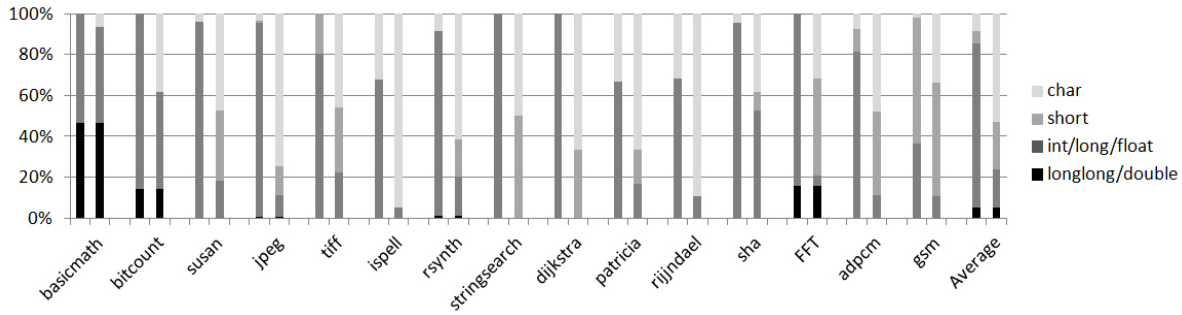


Figure 5.2: Change of variable type composition at the source-level

5.2.3 Memory accesses composition

Figure 5.3 shows the change of memory accesses composition between original programs and optimized programs when both programs are compiled with no optimization option. The left bar shows the percentage composition of memory accesses in the original program and the right bar shows the percentage composition of memory accesses in the optimized program out of total number of memory accesses in the original program. Although more than 50% of variables are reassigned to the best fit type, the change of memory access composition is relatively small in the optimized program. However, the change is still visible. Number of 4 byte type accesses is reduced from 83.95% to 62.13% out of total number of memory accesses in the original program. The same analysis is repeated when compiling with the -O3 option as shown in Figure 5.4. There still is a significant reduction in 4-byte sized accesses.

5.2.4 Total bytes of memory accesses

Considering the change of memory access composition, the total bytes of memory accesses should decrease unless the total number of memory accesses increase. Figure 5.5 shows the normalized percentage of total bytes to access memory in the optimized program compared to the one in the original program. The optimized programs have 15.28% and 4.83% less bytes to access memory than the original program on average with no optimization option and -O3 optimization option respectively.

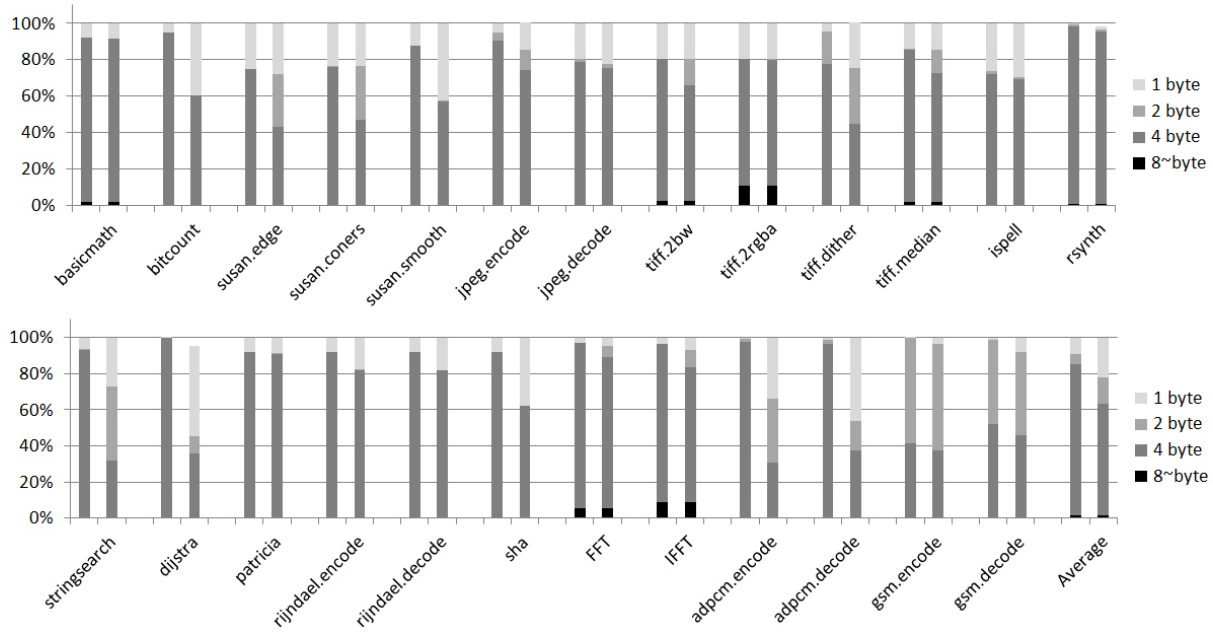


Figure 5.3: Change of memory access composition without optimization option

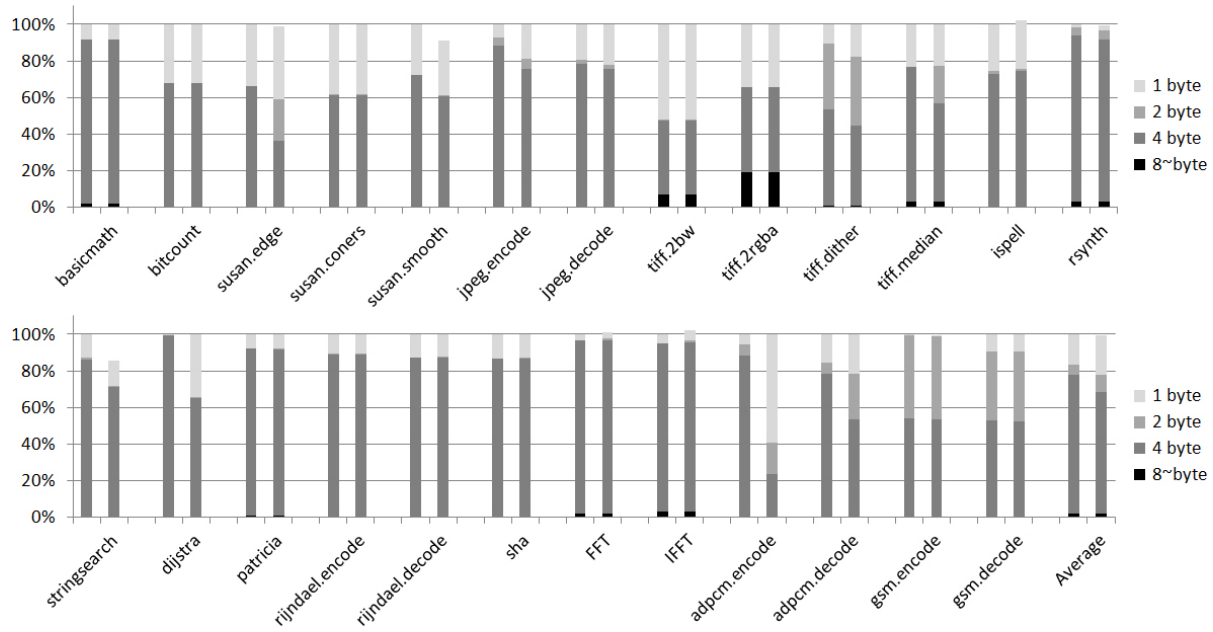
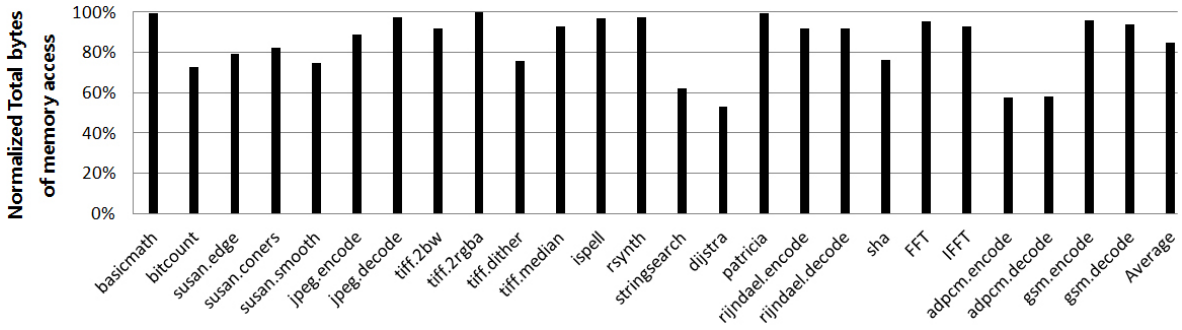
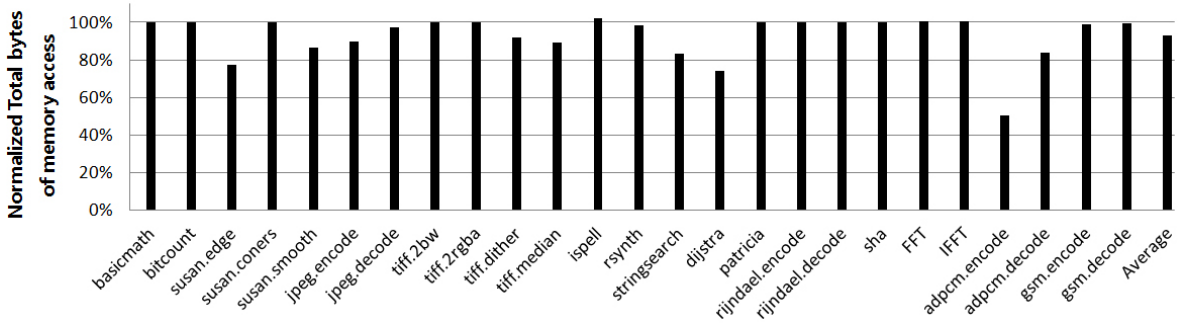


Figure 5.4: Change of memory access composition with -O3 optimization option



(a) No optimization option

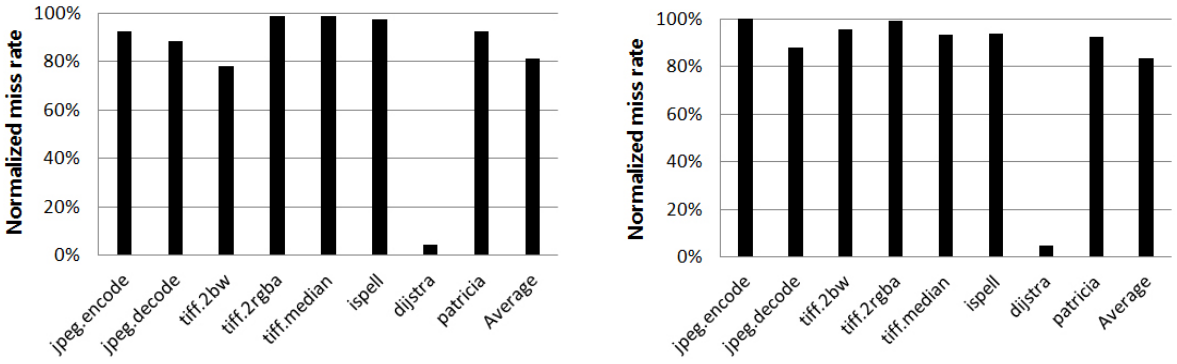


(b) -O3 optimization option

Figure 5.5: Total bytes of memory accesses

5.2.5 Cache miss rate

Since the benchmarks suites in MiBench are relatively small, some of the applications have almost no L1 cache misses. Therefore, 5 benchmarks and their inputs which have more than a 0.1% L1 cache miss rate were selected for a more detailed analysis. The percentage of L1 cache miss rate in the optimized program is shown in Figure 5.6. The optimized programs have 18.55% and 16.61% less miss rate on average in the L1 cache compared to the un-optimized code with the same compilation options. In the *dijstra* case, our optimizations eliminate more than 95% of the cache misses. In fact, there are no capacity misses in the optimized *dijstra* benchmark, so it is possible to infer that all data used in *dijstra* fits into the L1 cache after the optimization.



(a) No optimization option

(b) -O3 optimization option

Figure 5.6: L1 cache miss rate

Chapter 6

Conclusion

6.1 Conclusion

Dynamically deriving value-range invariants expands a programmer's ability to select a more efficient data structure. Considering the range of a variable, a programmer may be able to choose the best-fit type for the variable. Using the best-fit type for variables, the total bytes of memory accesses is reduced.

This thesis presents a new analysis framework for dynamically detecting value-range invariants at the source-level and describes a manual optimization scheme for assigning the best-fit type for a variable. In general, such optimizations are well known to programmers. However, The study shows that programs contain a significant amount of storage waste. By attacking waste uniformly and comprehensively throughout a program using automated analysis tools, significant storage savings are possible.

6.2 Future work

In the future, I plan to fully automate the code optimization process. This would not be a safe optimization, but it would make it easier for a programmer to benefit from our analysis tool. Furthermore, I will analyze a much wider range of programs to demonstrate the importance of this technique. In particular, I want to focus on programs with high L2 miss rates to demonstrate the importance of reducing storage costs throughout a program in order to relieve pressure on the last level cache.

REFERENCES

[1] Clang project. http://clang.llvm.org.

[2] Dinero IV. http://www.cs.wisc.edu/∼markhill/DineroIV.

[3] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *High-Performance Computer Architecture, Proceedings. Fifth International Symposium On*, pages 13–22, 1999.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Microarchitecture, Thirtieth Annual IEEE/ACM International Symposium on*, pages 259–269, 1997.

[5] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. *J. Instruction-Level Parallelism*, 1, 1999.

[6] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2–3):99–123, 2001.

[7] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[8] C. Euiyoung, B. Luca, G. DeMicheli, G. Luculli, and M. Carilli. Value-sensitive automatic code specialization for embedded software. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(9):1051–1067, 2002.

[9] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, 2001.

[10] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[11] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[12] M. A. Khan. Improving performance through deep value profiling and specialization with code transformation. *Computer Languages, Systems and Structures*, 37(4):193–203, 2011.

[13] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *SIGOPS Oper. Syst. Rev.*, 40(5):37–48, 2006.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[15] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.

[16] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, 2008.

[17] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 108–120, 2000.

[18] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Programming language design and implementation, Proceedings of the ACM SIGPLAN conference on*, pages 174–186, 2010.

[19] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 137–152. Springer Berlin / Heidelberg, 2002.

[20] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 269–280, 2004.