View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2004-13

2004-03-29

Static Analysis of Memory-Accessing Gestures in Java

Christopher R. Hill

We propose the notion of Java-program gestures that are composed of a series of memory-accessing instructions. By finding patterns in gestures whose execution can be atomic, we can load them in an intelligent memory controller. This process can improve performance of the Java Virtual Machine, decrease code footprint, and reduce power consumption in hardware. In this thesis we formally define a language of gestures and introduce a method of detecting them statically at compile-time. We introduce a simple heuristic for reducing the number of gestures that must be loaded into the memory controller and show that finding the minimum... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Hill, Christopher R., "Static Analysis of Memory-Accessing Gestures in Java" Report Number: WUCSE-2004-13 (2004). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/985

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/985

Static Analysis of Memory-Accessing Gestures in Java

Christopher R. Hill

Complete Abstract:

We propose the notion of Java-program gestures that are composed of a series of memory-accessing instructions. By finding patterns in gestures whose execution can be atomic, we can load them in an intelligent memory controller. This process can improve performance of the Java Virtual Machine, decrease code footprint, and reduce power consumption in hardware. In this thesis we formally define a language of gestures and introduce a method of detecting them statically at compile-time. We introduce a simple heuristic for reducing the number of gestures that must be loaded into the memory controller and show that finding the minimum number is NP-Complete. We profile the performance of this algorithm extensively on a set of Java benchmarks.

Department of Computer Science & Engineering

2004-13

Static Analysis of Memory-Accessing Gestures in Java - Master's Thesis, May 2004

Authors: Hill, Christopher R.

March 29, 2004

Abstract: We propose the notion of Java-program gestures that are composed of a series of memory-accessing instructions. By finding patterns in gestures whose execution can be atomic, we can load them in an intelligent memory controller. This process can improve performance of the Java Virtual Machine, decrease code footprint, and reduce power consumption in hardware.

In this thesis we formally define a language of gestures and introduce a method of detecting them statically at compile-time. We introduce a simple heuristic for reducing the number of gestures that must be loaded into the memory controller and show that finding the minimum number is NP-Complete. We profile the performance of this algorithm extensively on a set of Java benchmarks.

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Short Title: Static Analysis of Java Gestures Hill, M.Sc. 2004

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

STATIC ANALYSIS OF MEMORY-ACCESSING GESTURES IN JAVA

by

Christopher R. Hill B.S. Applied Science

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

STATIC ANALYSIS OF MEMORY-ACCESSING GESTURES IN JAVA

by Christopher R. Hill

ADVISOR: Dr. Ron K. Cytron

May, 2004
Saint Louis, Missouri

We propose the notion of Java-program gestures that are composed of a series of memory-accessing instructions. By finding patterns in gestures whose execution can be atomic, we can load them in an intelligent memory controller. This process can improve performance of the Java Virtual Machine, decrease code footprint, and reduce power consumption in hardware.

In this thesis we formally define a language of gestures and introduce a method of detecting them statically at compile-time. We introduce a simple heuristic for reducing the number of gestures that must be loaded into the memory controller and show that finding the minimum number is NP-Complete. We profile the performance of this algorithm extensively on a set of Java benchmarks.

For My Friends and Family

Contents

List of Figures . . . . . **vi**

Acknowledgments . . . . . **viii**

1 Introduction **1**

1.1 What is a Gesture? 1

1.2 Definition of a Language for Gestures 4

1.3 Current Handling of Gestures 5

1.4 Definition of the Language of Macros 6

1.5 Summary of Results 8

2 Complexity Analysis of the Optimization Problem **10**

2.1 Problem Generalization 10

2.2 Problem Statements 12

2.3 The NP-Completeness Proof Model 12

2.4 Theorem 1 Proof 13

2.4.1 Theorem 13

2.4.2 Verifiability 13

2.4.3 Reduction from Subset-Sum 13

2.4.4 Forward Proof 15

2.4.5 Reverse Proof 17

2.4.6 Example 18

2.5 Theorem 2 Proof 19

2.6 Theorem 3 Proof 19

2.7 Theorem 4 Proof 20

2.8 Additional Conjectures 20

3 Greedy Heuristic for Field Reordering **22**
3.1 Concepts 22
3.2 The Algorithm 23
3.3 Counterexample 24

4 Scavenge Design and Implementation **27**
4.1 Design 27
4.1.1 Path Parsing 27
4.1.2 The Reorder Package 28
4.2 Usage 30

5 The Seekr Program **32**
5.1 Design Considerations 32
5.1.1 Program Motivation 32
5.1.2 Motivation for Using Clazzer 32
5.2 Implementation Considerations 35
5.2.1 Building the Type Table 35
5.2.2 Building the Instruction Table 35
5.2.3 Counting Macros 36
5.2.4 The Strategy Pattern 36
5.2.5 Different Strategies 36
5.3 Usage 37

6 The MacroSimulator Program **39**
6.1 Input: The .sprob and .sin files 39
6.2 Using Seekr to Generate Simulator Input 40
6.3 The ScavengeModel Class 41
6.4 The Visualizer 41
6.5 Usage 43

7 Experiments **45**
7.1 The Java Benchmarks 45
7.2 Finding Indirection Chains with Javap 46
7.3 Benchmark Statistics 46
7.4 Seekr Timing Analysis 47
7.5 Indirection Chain Execution Timing 47

7.6 Measuring Idea Feasibility with MacroSimulator 48
7.7 Exhaustive Reordering 48
7.7.1 The Permute Stage 49
7.7.2 The Combine Stage 49
7.8 Random Reordering 50
7.9 Greedy Reordering using Scavenge 50
7.10 Greedy Reordering using Seekr 51
7.11 Results 51
7.11.1 Indirection Chain Count Results 51
7.11.2 Timing Results 53
7.11.3 Field Reordering Results 56
7.12 Experimental Conclusions 62

8 Conclusion **63**
8.1 Future Work 64
8.2 Final Thoughts 64

References **66**

Vita **68**

List of Figures

1.1 Stack Operation of a Double Indirection 3
1.2 FSA that Accepts Our Gesture Language 5
1.3 Execution of a Double Indirection 6
1.4 Execution of a Triple Indirection Ending in a Putfield Instruction 7
1.5 Execution of a Double Indirection with a PIM 8
1.6 Execution of a Triple Indirection Ending in a Putfield Instruction with a PIM 9

2.1 Pictorial example of NP reduction 18

3.1 Example Field Alignment 23
3.2 Example Instruction List (p, q, r, s, and t are instances of Foo objects) . 23
3.3 Example Frequency Chart 24
3.4 Example Field Alignment after Algorithm Execution 24
3.5 Counterexample Field Alignment 25
3.6 Counterexample Instruction List 25
3.7 Counterexample Frequency Chart 26

5.1 Two Getfield Chains: (a) one possible data flow path (b) two possible paths 34

6.1 The Simulator Main Screen 41
6.2 The Hardware Options Screen 42
6.3 The Software Options Screen 43

7.1 Gesture Counts using Javap -c 52
7.2 Benchmark Statistics 53
7.3 Benchmark Size 54
7.4 Seekr Timing Profile 55
7.5 Timing Analysis of Benchmarks 55
7.6 Execution Time of an Indirection Chain vs. Chain Length 56

7.7 Indirection Chain Timing With and Without Cache 57
7.8 Time Savings of the Macro Strategy from MacroSimulator 57
7.9 Graph of Simulated Execution Time With and Without Macros 58
7.10 Resulting Bounds from Exhaustive Ordering 58
7.11 Mean and Minimum from 25 Random Trials for Each Benchmark 60
7.12 Performance Benefit of the Greedy Algorithm Using Scavenge (OOB is Out of Box) 61
7.13 Algorithm Comparison in Seekr 61
7.14 Graph of Greedy Performance Against Measured Bounds 62

Acknowledgments

Thanks to NSF, whose funding through the ITR-0081214 grant allowed me to maintain my lavish lifestyle of opulence while completing this work.

Special thanks goes out to Dr. Ron Cytron, whose intelligence, pastries, and hockey playing inspired me to remain in academia for a few more years before bolting to the land of the new economy, signing bonuses, and stock options. They'll all be waiting for me when I'm done, right?

Thanks to Lucas Fox for being the brains - and height - of this operation. He may have proved that my greedy heuristic was nonoptimal, but he more than made up for it with his great insights, ability to mathematicize the most abstract of ideas, and ability to make layups. Because of him, I will always be a "poor man's Lucas".

Thanks to all my colleagues in the DOC lab who I was fortunate enough to work with, the Monday morning basketball folks, and the softball team. All work and no play would have made me a dull boy.

And most of all, thanks to my parents, who are responsible for the work ethic and values that have helped me to succeed in everything.

Christopher R. Hill

Washington University in Saint Louis
May, 2004

Chapter 1

Introduction

During the past 7 years, Java has gone from the "future of the Internet" to an online afterthought with the introduction of Flash, but the language is making news once again as a possible language of choice for cell phones and other handheld devices. Nokia already uses Java, citing the ease in which third-party companies can develop applications in the language as a motivating factor [13], and as cell phones expand from voice communication devices to portals for accessing the web and playing games, Java could move even more into the forefront.

When a heavyweight language like Java is crammed into a sleek package such as a mobile phone, speed and memory footprint are much more important than when running Java on a personal computer. As phones get smaller and more complex, more and more instructions must run efficiently using the available memory.

In this thesis, we introduce one approach that marries software intuition with hardware efficiency to achieve this goal.

1.1 What is a Gesture?

The Java programming language is structured so as to allow programs written in one generic syntax to be run on any platform. Source files written by a programmer are compiled into `.class` files that are interpreted at runtime by a **Java Virtual Machine** (JVM), which is a platform-specific implementation. Therefore, as long as you have a JVM implementation for your platform, you can run any compiled Java program in that environment. This functionality has been called "write once, run anywhere" [1].

A programmer using the Java language typically writes `.java` files (later compiled into `.class` files) called classes that interact with other user-defined classes as well as pre-built ones in the various Java libraries. Together, these `.class` files form a Java program.

A `.class` file is composed of the following parts:

methods contain the instructions that are compiled into opcodes and executed by the JVM.

fields are variables that exist in every instance of the class.

One example of an instruction that reads the value of a field from main memory is the `getfield` opcode. This opcode takes the top address off the stack and returns from main memory the contents of a specified field of the object that is referenced by the address taken from the stack [8].

For example, the following code loads the address of an instantiated Foo object from register 4 and then calls `getfield` on constant pool entry #68, which is a field named a of type Bar:

```
aload 4
getfield #68 <Field Bar a>
```

Because fields can themselves contain references to other objects, `getfield` opcodes can be executed sequentially to obtain the value from a field in an object that is referenced from another object. We refer to consecutive `getfield` instructions as **indirection chains**, because when each individual opcode returns its resulting address to the stack, that address is used as the starting point for the next `getfield` opcode. The following example shows an indirection chain of length two, also called a **double indirection**:

```
aload 4
getfield #68 <Field Bar a>
getfield #72 <Field Foo f>
```

In this case, `f` is a field of type `Foo` that is found in an instance of a Bar object. The stack operation during execution of these opcodes is shown in Figure 1.1. The `aload` opcode loads the initial `Foo` object reference onto the stack. The first `getfield` pops that reference and pushes the contents of the specified field, which is a reference

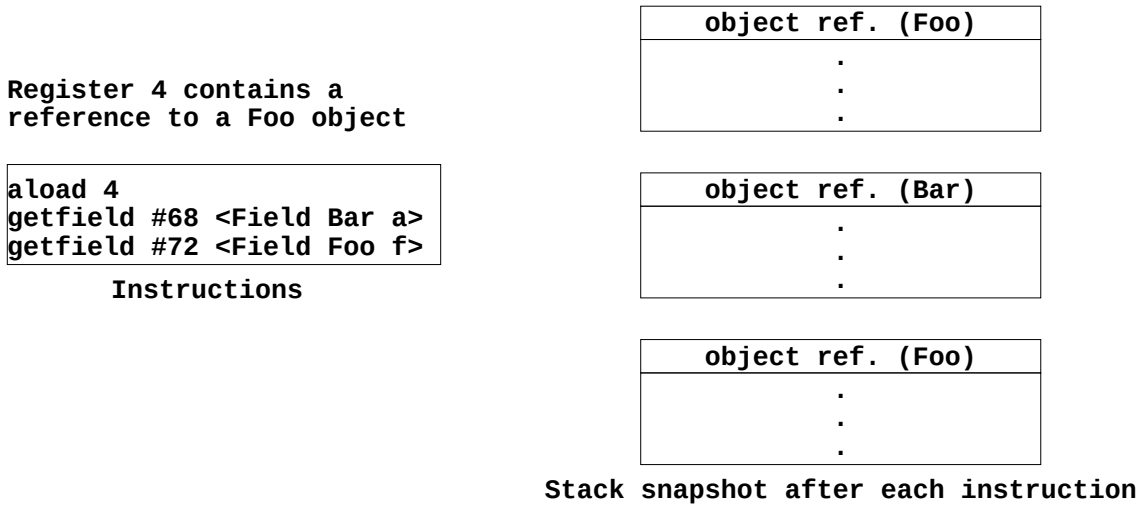

Figure 1.1: Stack Operation of a Double Indirection

to a `Bar` object. Finally, the second `getfield` pops this reference and pushes the contents of the last field specified, which is a reference to another `Foo` object. This reference is left on the top of the stack for future instructions to use.

There are other instructions that access memory besides `getfield`, though, with the most prevalent being `putfield`. This opcode assigns the value found on the top of the operand stack to the field specified by the opcode in the class reference that is directly below it on the stack. For example, the following code would put the number 5 into integer field `i` of a newly-created `Foo` object:

```
new #9 <Class Foo>
dup
invokespecial #14 <Method Foo()>
astore_1
aload_1
iconst_5
putfield #18 <Field int i>
```

The `invokespecial` opcode calls the constructor of Foo, then the reference to that class is stored in register 1 and then loaded to the top of the stack. Next the integer 5 is pushed onto the stack, then the `putfield` opcode is called on field `i`.

A `putfield` can also be a part of an indirection chain, as shown here:

```
new #9 <Class Foo>
dup
invokespecial #14 <Method Foo()>
astore_1
aload_1
getfield #18 <Field Bar a>
getfield #22 <Field Foo f>
iconst_5
putfield #28 <Field int i>
```

This machine code is generated from a source file that simply constructs a `Foo` object called `foo` and then does the assignment `foo.a.f.i = 5`. Here the 3 memory-accessing opcodes are still part of a chain even though they do not directly follow each other in the machine code. This is because the result of the first two `getfield` operations is not used on its own. It is just an intermediary used by the `putfield` operator. Because this chain contains three memory-accessing opcodes, it is classified as a **triple indirection**.

1.2 Definition of a Language for Gestures

We define a **gesture**, similar to a **superoperator** [12], as a series of logically-consecutive opcodes. Valid gestures can be defined using a language that includes any sequence of `getfield` opcodes that ends with a `getfield`, `putfield`, or `putstatic` opcode. In addition, the sequence can optionally begin with a `getstatic` opcode. A gesture is a **subsequence** of the instruction set. In other words, these instructions do not have to follow each other consecutively in the code, but the values returned by the intermediary memory-accessing opcodes cannot be used anywhere else except by the next memory-accessing opcode.

The language of consecutive gestures can therefore be defined as:

$$(\texttt{getfield} \mid \texttt{getstatic})(\texttt{getfield})^*(\texttt{getfield} \mid \texttt{putfield} \mid \texttt{putstatic})$$

Note that once again we only accept chains of length two or greater. Another definition uses a **Finite State Automata** (FSA) that accepts only valid gestures as defined by this language. The FSA is shown in Figure 1.2.

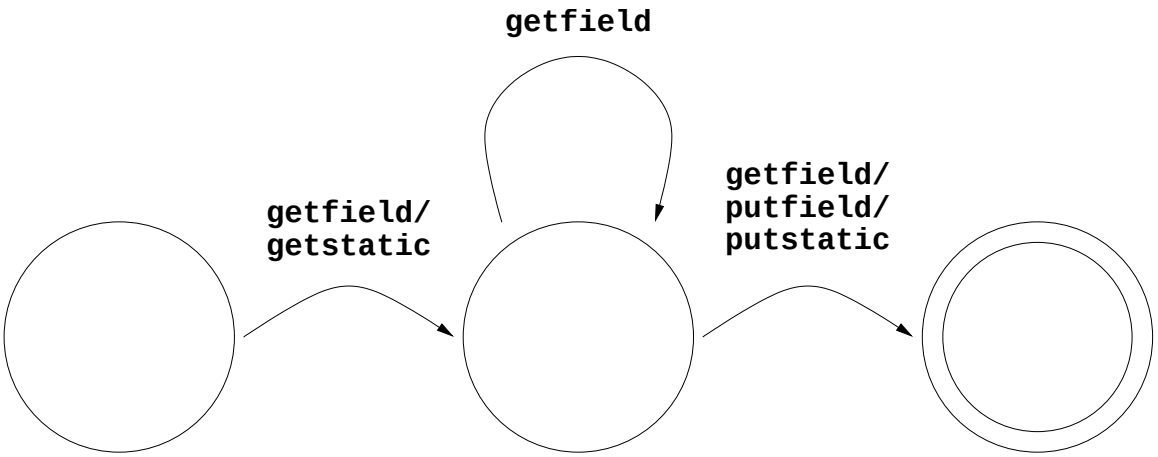

Figure 1.2: FSA that Accepts Our Gesture Language

1.3 Current Handling of Gestures

Although superficially simple, repetitive indirection from memory has some unnecessary overhead. When a CPU encounters a `getfield` opcode, it retrieves the address from the stack and then sends it over to memory. The memory controller then finds the target address and sends it back to the CPU. Then CPU executes the next `getfield` opcode and sends the address it just got, plus some offset, right back to memory. This process is repeated until the final `getfield` opcode. After this instruction is executed, the desired value is at the top of the stack on the CPU. The block diagram shown in Figure 1.3 demonstrates this process for the double indirection example.

However, only the final value is of ultimate interest at the CPU, since the other values returned from memory are just intermediate addresses in the longer computation. In the example above, the CPU receives the address of `Bar a` even though the goal of that series of instructions is to find the value of Foo f. After fetching the value of `Foo f`, the address of `Bar a` is popped off the stack, so it is unused by any other instruction.

Figure 1.4 shows an example of the same process for a triple indirection ending in a `putfield`. The operation shown is `foo.a.f.i = x`, where `x` is some integer (int). Here the excessive use of the memory bus for values that the CPU does not use can be clearly seen. The CPU receives the memory addresses of `a`, `f`, and `i` even though they are not used anywhere else in the program.

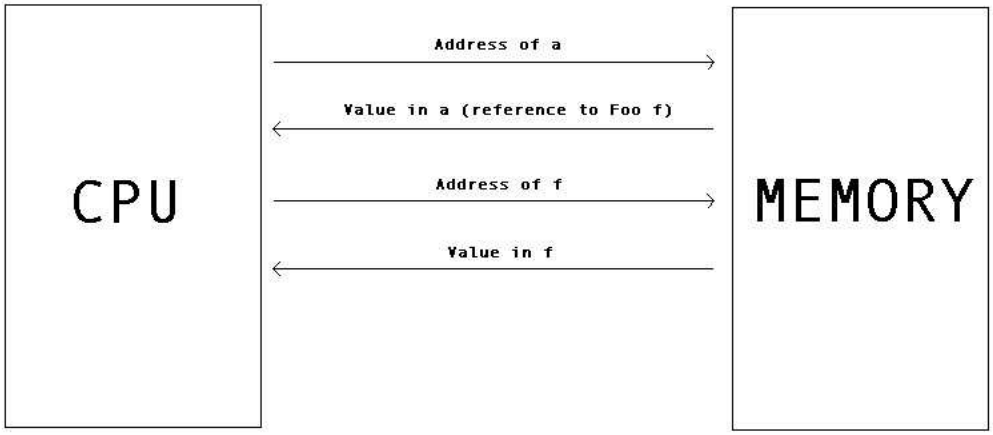

Figure 1.3: Execution of a Double Indirection

If we could somehow keep the intermediate information in an intelligent memory controller, we could avoid accessing the CPU each time. We call this type of smart memory chip–which could be implemented using a **Field-Programmable Gate Array** (FPGA)–**Intelligent RAM** (IRAM) or a **Processor in Memory** (PIM) [7, 11].

Using a PIM could theoretically benefit running a Java program as follows:

- **Fewer Memory Bus Cycles:** We would reduce the number of trips back and forth to main memory; therefore, we correspondingly reduce memory bus usage.
- **Less Power Consumption:** Because we would not be using the memory bus as often, we would be charging it less frequently as well.
- **Better CPU Throughput:** We would also free up CPU cycles on a multi-threaded system for other processes.

1.4 Definition of the Language of Macros

Gestures could be encapsulated on the PIM as **macros**, which tell memory to fetch the contents of a certain address, treat the contents of that address as an address, fetch the contents of the kth field of the object found at that address, and continue as

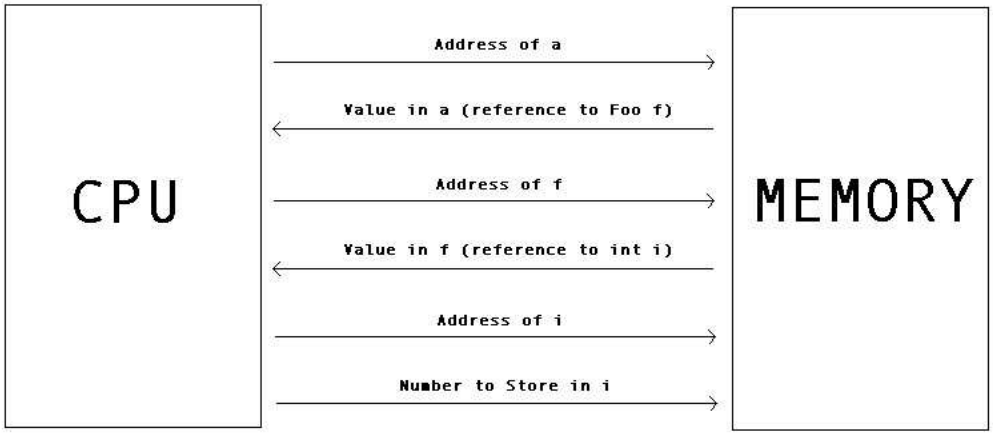

Figure 1.4: Execution of a Triple Indirection Ending in a Putfield Instruction

long as is necessary. The macros can then be reduced to a sequence of field numbers, with the starting address provided by the CPU upon calling the macro. If there is a macro that can execute each gesture in a program, that program is said to be **covered** by the set of macros.

For example, if class `Foo` has field `a` as its first field and class `Bar` has field `f` as its second field, the macro that would cover this gesture would be the ordered tuple <1, 2>. This macro would be stored on the PIM, and whenever the application encounters this gesture it would send an identifier for the corresponding macro and the address of the `Foo` object to the PIM.

In general, a macro can be described using the following regular language:

$$< \{k, \}^{+} k >, k \in N$$

Note that a macro must have at least two indices, since we would not expect any benefit from finding macros of length one.

There is some overhead in using this process, since the gestures must be located in the `.class` files and replaced with new opcodes sometime before runtime and the macros must be stored on the PIM prior to execution. Therefore, the idea is most

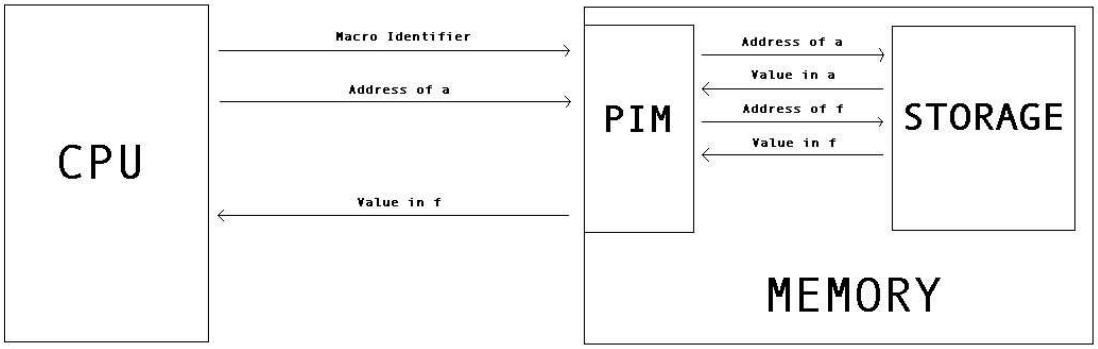

Figure 1.5: Execution of a Double Indirection with a PIM

profitable if `.class` files contain either very long gestures or a large number of small gestures that are executed frequently by the JVM.

The benefits of such a system are shown in Figure 1.5 using the same double indirection example as before. Notice that while the PIM is doing the indirections, the CPU is not utilizing the memory bus and is free to execute other processes.

The back-and-forth motion is still there, but now it is totally encapsulated in memory. Regardless of the length of a `getfield` indirection chain, we send two items to memory and receive one in return. The first item we send is the macro identifier that tells the PIM which macro to use. The second is the address of the initial object as a starting point for the macro. Once the PIM finishes executing the macro, it returns the desired value.

For chains that end in a `putfield`, we send three values and receive none in return. As before, we send the macro identifier and the starting address, but here we also send over the value to store in the target field. Figure 1.6 shows this process in detail for a triple indirection that stores a number into an integer field.

1.5 Summary of Results

This thesis will the summarize the results we got by performing the following steps:

- Determine the extent to which chains of memory-accessing instructions exist in Java programs.

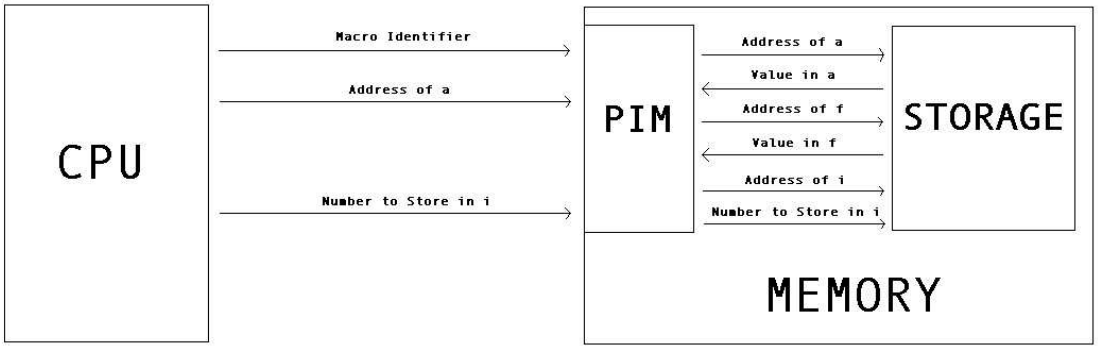

Figure 1.6: Execution of a Triple Indirection Ending in a Putfield Instruction with a PIM

- Model the timing of instruction chains at runtime.
- Develop an effective algorithm for minimizing the number of macros needed to cover a program.
- Measure the effectiveness of our algorithm and the PIM idea in general in a simulation of the real world.

In the next two chapters we explore the concept of reordering fields in `.class` files in an effort to reduce the number of macros we need to cover a program.

Chapter 2

Complexity Analysis of the Optimization Problem

While simply finding gestures and translating them into macros is an effective way of reducing source code footprint, it does not necessarily guarantee any overall memory savings. Macro information must still be stored in main memory, so if we can limit the number of macros needed to cover all dereferencing instructions in a program, then we can limit the amount of memory overhead that this system requires.

One effective way to reduce the number of macros while maintaining program behavior is to permute the order of the fields in each Java `.class` file. Because macros are just numbers that represent position of fields in their respective class files, then if we have the macro <`x`,`y`> it would be beneficial for us to put as many fields that are part of dereferencing instructions as possible in the `x`th and `y`th positions of their class files.

For any program, there is some ideal permutation of fields that will fit all dereferencing instructions into the smallest possible number of macros, but how hard is it to find this perfect field alignment? In this chapter we prove that this problem is NP-complete. We begin with a formal definition of our problem and then present a series of proofs.

2.1 Problem Generalization

Given:

- A positive integer k that is the exact length of all gestures.

- A program $P = (T, FT, I)$ where

  T is a set of types referenced by the program.

  FT is a mapping

  $$FT : (T \times N) \rightarrow T$$

  For a given type $t \in T$, $FT(t, n)$ is the type of the nth field of t according to the layout of type T.

  $I \subseteq \{(t, m) \mid t \in T, m \in N^k\}$ is the multiset of the program's instructions. Here, t is the type of an instruction's first reference. The vector m provides successive offsets used for dereferencing.

- We define a vector of types τ((t,m)) for each instruction $g \in I$ as follows:

  $$\begin{aligned} \tau_0 &= t \\ \tau_i &= FT(\tau_{i-1}, m_i), 1 \leq i \leq k \end{aligned}$$

  Thus, τ_j is type of the jth indirection for instruction g.

- A permutation ρ permutes the fields of a type as follows:

  $$\rho : (T \times N) \rightarrow N$$

- $\lambda((t, m), \rho)$ represents the effect of a permutation ρ on instruction (t, m) as follows

  $$\lambda((t, m), \rho) = (t, m')$$

  where $m'_i = \rho(\tau_i((t, m)), 0 \leq i \leq k$. The extension of λ to a *set* of instructions is straightforward.

- $M \subseteq N^k$ is a set of macros available to the program.

- A positive integer μ that bounds the size of M.

- For a subset of instructions $S \subseteq I$ and a set of macros M, let $C(S, M)$ be the set of instructions in S covered by macro set M:

  $$C(S, M) = \{s \in S \mid \exists t \ \exists m \ (t, m) = s, \ m \in M\}$$

- A positive integer β that bounds the size of the above set.

2.2 Problem Statements

We next seek to determine the complexity of finding a permutation that results in the fewest number of macros to cover an instruction multiset. Our approach is to consider a sequence of simpler decision problems as follows.

$D1(P, \beta)$ $(\exists\ S \subseteq I)$ $(\exists\ \rho)$ $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| = 1$

$D2(P, \beta, \mu)$ $(\exists\ S \subseteq I)$ $(\exists\ \rho)$ $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| = \mu$

$D3(P, \beta, \mu)$ $(\exists\ S \subseteq I)$ $(\exists\ \rho)$ $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| \leq \mu$

Problem $D3(P, \mu)$ determines whether μ macros suffice to cover β instructions in program P. The optimization form of this problem is to find

$$Opt(P, \beta) = \min_{n \in N} D3(P, \beta, n)$$

2.3 The NP-Completeness Proof Model

We will use an NP-Completeness reduction to show that our problem is NP-Hard. Before presenting proof itself, we will first explain how the NP-Completeness proof model works.

There are a number of complexity classes for categorizing problems. **P** represents the class of problems that can be solved in polynomial time, whereas **NP** represents the set of problems that can be solved on a nondeterministic machine in polynomial time.

To prove that a problem is **NP-Complete**, that is, that there does not exist an algorithm that solves the problem in polynomial time unless P = NP, it suffices to perform the following two steps:

- Show that if we are given a solution and told that it is correct, we can verify both validity and correctness in polynomial time; that is, the problem is in NP.
- Prove that every other NP-Complete problem can be reduced to our problem in polynomial time and space, in other words that the problem is NP-Hard.

Typically, the second part is accomplished by reducing a known NP-Complete problem to the new problem in polynomial time and space, since one property of the set of NP-Complete problems is that they all reduce to each other in polynomial time and space.

A reduction is shown by constructing a specific instance of our problem from an arbitrary instance of a known NP-Complete problem and showing that there is a polynomial-time solution to the NP-Complete problem if and only if there is a polynomial-time solution to our problem [5].

2.4 Theorem 1 Proof

2.4.1 Theorem

$D1(P, \beta)$: $(\exists\ S \subseteq I)\ (\exists\ \rho)\ |C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| = 1$ is NP-complete when the number of fields per type is greater than 1 for any type in T.

2.4.2 Verifiability

We can reorder the instruction set I with a given ρ in polynomial time because for each instruction we can use ρ along with the instruction's t and m to find $\lambda(t, m)$ as described above in linear time $(\Theta(|I| * k))$. We can then count the number of instructions covered by each macro in M in linear time $(\Theta(|I| * \mu))$ and sum up the total number of instructions covered in linear time $(\Theta(|I|))$. Thus, the total verification time is $\Theta(|I| * k + |I| * \mu + |I|) = \Theta(|I| * (k + \mu + 1))$, which is polynomial.

2.4.3 Reduction from Subset-Sum

To prove that the problem is NP-complete, we will show that the Subset Sum problem reduces to it.

Subset Sum asks whether a finite set of sized elements A contains any subset of elements A' whose sizes sum up to a positive integer B. It was shown to be NP-complete with a transformation from Partition by Karp [6]

First we define an instance of problem 1 in terms of a subset sum problem. In subset sum we are given a finite set A, a size $s(a) \in Z^+$ for each $a \in A$, and a positive integer B.

Let's consider an arbitrary instance of subset sum using the notation above where we are given our set A and our integer B.

We can construct the following specific instance of our problem using the following constraints:

- Let $\beta = B$.
- Let the variable n be $|A|$.
- Let k be $\log_2 n$.
- Let our program $P = (T, FT, I)$ be defined as:

  Assign some arbitrary ordering $a_1, a_2, \ldots, a_n$ to the elements in A.

  Let T be a set of $n \log_2 n$ unique types, with each type containing exactly two fields (0 and 1). Let the set be enumerated as follows:

  $$T = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k} \{(t_{i,j})\}$$

  Let FT be defined as:

  $$t_{i,j} \times \{0, 1\} \ = t_{i,j+1}$$

  It follows from this construction of FT that each unique instruction will reference a series of types $t_{i,1}, t_{i,2}, t_{i,3}, ... \in T$ such that each type is referenced by by at most one unique instruction.

  Define function $enc(i)$ which returns the vector of binary digits that represent the base 2 numeral for i:

  $$enc : I \rightarrow \{0, 1\}^k$$

  In our problem $enc(i)$ would represent m, the sequence of field numbers being accessed in each successive indirection step of the gesture. A 0 would indicate the instruction is referencing the first field in a type and a 1 would indicate the instruction is referencing the second field in a type.

This will allow us to create n unique binary instructions. That is, we ensure there are enough binary digits to create an instruction that references a series of unique types and has a unique m, for each starting type $(t_{1,i})$.

Let I be the multiset of ordered pairs:

$$I = \bigcup_{i=1}^{n} \bigcup_{c=1}^{s(a_i)} \{(t_{i,0}, enc(i))\}$$

For each $a \in A$, we create a unique instruction (n total) and then "clone" each one $s(a)$ times.

From our definition of FT, it follows that our type vector τ for any instruction $(t_{i,1}, enc(i))$ will be $[t_{i,1}, t_{i,2}, \ldots, t_{i,k}]$

- Let $|M| = 1$, we want to use only one macro.

2.4.4 Forward Proof

First, we need to show that a solution to the subset sum problem implies a solution to our problem:

A solution to the subset sum problem exists when

$$\exists A' \subseteq A \text{ where } \sum_{a \in A'} s(a) = B$$

We then need to show $\exists S$ and $\exists \rho$ such that $|C(\lambda(S,\ \rho),\ M)| = \beta$ and $|M| = 1$.

- Choose some subset S $\subseteq$ I such that:

$$S = \bigcup_{a \in A'} \bigcup_{c=1}^{s(a)} \{(t_{a,1}, enc(a))\}$$

Note that the size of S is B.

- In order to cover exactly $\beta = B$ instructions with M we need to create a ρ that permutes I in such a way that all the instructions in S are covered by the single element z in M, and no other instructions in I are covered by z. In other words, we need:

$$(\exists z)\ (\forall (t,m) \in S\ \lambda((t,m),\rho) = (t,z),\ \forall (t,m) \in I - S\ \lambda((t,m),\rho) \neq (t,z))$$

Here we will let z be a vector of all 0's (0^k).

Because

$$\bigcap_{\forall i \in I} \tau(i) = \emptyset$$

(that is, the types referenced by each instruction in I are disjoint), we can define their permutations independently.

We define ρ as:

$$\rho(t_{i,j} \times l) = \begin{cases} l & \text{if } (t_{i,1}, enc(i)) \notin S \\ 0 & \text{if } (t_{i,1}, enc(i)) \in S,\ l = enc(i)_j \\ 1 & \text{if } (t_{i,1}, enc(i)) \in S,\ l \neq enc(i)_j \end{cases}$$

$$1 \leq i \leq n, 1 \leq j \leq k, l \in \{0,1\}$$

where j is the index (left to right) into vector $enc(i)$.

- Then

$$\forall (t,m) \in S\ \ \lambda((t,m),\rho) = (t, 0^k)$$

that is, every element and only those elements in $\lambda(S,\rho)$ are of the form $(t_{a,1}, 0^k), a \in A'$. So, by our definition of C:

$$|C(\lambda(S,\ \rho),\ M)| =\ B\ =\ \beta$$

Thus, there exists a S and ρ as defined above such that $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| = 1$.

2.4.5 Reverse Proof

Then, going in the other direction, we need to show that a solution to our problem implies a solution to the subset sum problem.

We want to show that whenever

$$\exists S \subseteq I,\ \exists \rho\ |C(\lambda(S,\rho), M)| = \beta,\ |M| = 1$$

then there exists A, A', and $s(a)$ such that

$$A' \subseteq A, \sum_{a \in A'} s(a) = B$$

- Let $A = \{i \in N \mid (t_{i,1}, enc(i)) \in I\}$.
- Let the weight of each $a \in A$, $s(a)$, be the number of times $(t_{i,1}, enc(i))$ appears in multiset I.
- Let $A' = \{i \in N \mid (t_{i,1}, enc(i)) \in S\}$.

Note that by this construction, S is equivalent to:

$$\bigcup_{a \in A'} \bigcup_{c=1}^{s(a)} \{(t_{i,0}, enc(i))\}$$

The cardinality of this set is β, and we have defined that $B = \beta$ in our problem definition. Therefore, we simply need to show that the cardinality of S is equivalent to $\sum_{a \in A'} s(a)$. We can determine the total number of elements in set S in terms of A' and $s(a)$ through summation because it is a *multiset* union. The derivation is as follows:

$$\begin{aligned} B &= \beta \\ &= |\bigcup_{a \in A'} \bigcup_{c=1}^{s(a)} \{(t_{i,0}, enc(i))\}| \\ &= \sum_{a \in A'} |\bigcup_{c=1}^{s(a)} \{(t_{i,0}, enc(i))\}| \\ &= \sum_{a \in A'} s(a) \end{aligned}$$

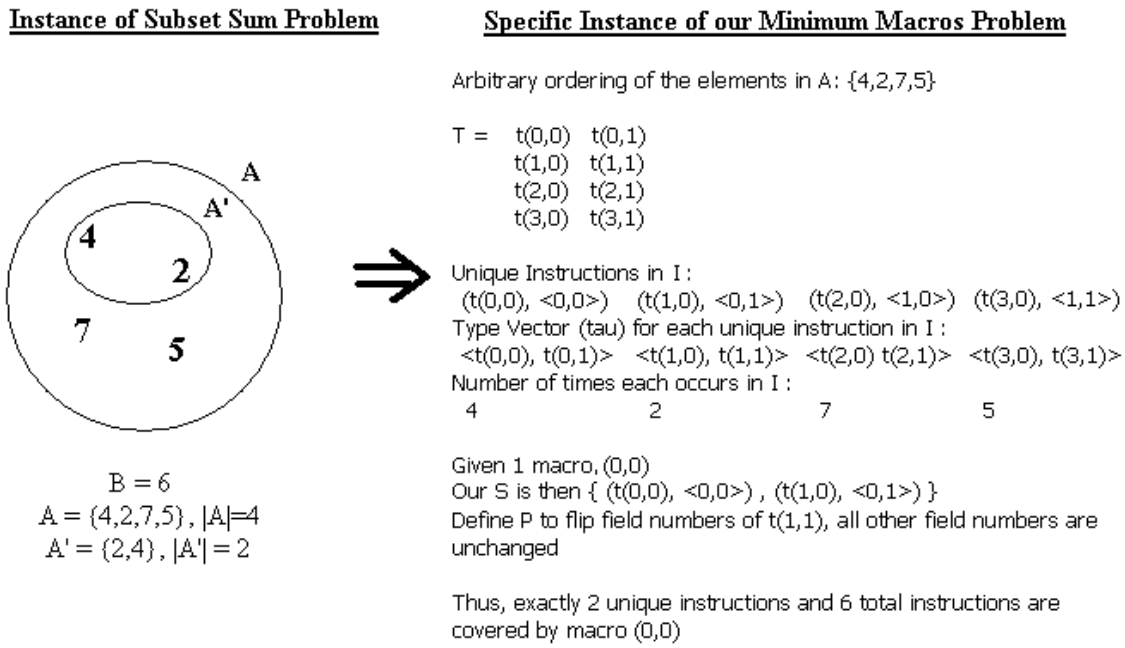

Figure 2.1: Pictorial example of NP reduction

Which means a solution to the subset problem exists.

Since we have now proven that a solution to the subset sum problem implies a solution to our problem and that a solution to our problem implies a solution to the subset sum problem, we can state that an instance of our problem is equivalent to subset sum and therefore NP-Complete. ■

2.4.6 Example

Because we have a unique set of "cloned" instructions for each type, no two groups of "cloned" instructions are be covered by the same macro unless we permute the fields of their referenced types. Because we have restricted S to include instructions that reference all different types, we can change ρ with the assurance that it will impact only one indirection in one instruction (and its clones).

These two properties give us the ability to cover all, some, or none of the instructions with our one macro. This can be done by simply flipping or not flipping the order of all types referenced by instructions that we want to be covered so that they conform to our macro.

For example, if the instruction we wanted to cover was "10", its corresponding type vector τ was <foo bar baz>, and our macro was "00"; we would need to flip the fields in foo but not baz or bar. However, the one restriction we have is that if we cover one instruction we are also covering all its clones. Because the number of clones is taken directly from the subset sum problem, we can only cover instructions in groups equal to $s(a)$ for some $a \in A$. This example is shown in Figure 2.1.

Therefore, by solving our problem we would also be solving the corresponding subset sum problem. If we can cover 5 instructions keeping in mind the group restraints, then we can also find a subset of A that sums up to 5. Likewise if we can find a subset of A that sums up to 5, that must mean that there are some number of groups of instructions that consist of 5 total instructions, and because we have the freedom to cover these groups we have solved our problem.

2.5 Theorem 2 Proof

Theorem: $D2(P, \beta, \mu)$ $(\exists\ S \subseteq I)$ $(\exists\ \rho)$ $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| = \mu$ is NP-complete.

This problem is simply D1 extended to multiple macros rather than restricting ourselves to just one. The rules for construction still remain the same, though, and this problem contains Problem 1 as a specific instance of it. By restriction, our problem above is also NP-complete when $|M| \geq 1$ because it is NP-complete when $|M| = 1$. ■

2.6 Theorem 3 Proof

Theorem: $D3(P, \beta, \mu)$ $(\exists\ S \subseteq I)$ $(\exists\ \rho)$ $|C(\lambda(S,\ \rho),\ M)| = \beta,\ |M| \leq \mu$ is NP-complete.

This problem is a more general version of the question in Problem 2, which contains this problem as a specific instance. Because it is NP-complete to find the ρ where β instructions are covered by exactly μ macros, it is therefore at least as complex to find the ρ where β instructions are covered by μ macros or less. By restriction, this problem is also NP-complete. ■

2.7 Theorem 4 Proof

Theorem: Finding $Opt(P, \beta)$ where $Opt(P, \beta) = \min_{n \in N} D3(P, n)$ is NP-complete.

Here we are doing the same thing we are doing in problem 3 except we not only want a ρ to cover $\leq \mu$ macros, but we want that to be the absolute minimum number of macros that could possibly cover the β instructions. This problem is NP-hard, because we could only solve this problem if we could solve problem 3, but it is not NP-complete.

For a problem to be NP-complete, as described in § 2.3, it must not only extend from an NP-complete problem but also be verifiable in polynomial time. If we are given a solution to this problem, we can easily check that it is a valid solution in polynomial time, but there is no known way to determine whether that solution is the minimum solution without checking all other valid solutions. Therefore problem 4 is NP-hard. ■

2.8 Additional Conjectures

We have shown that finding the minimum number of macros needed to cover *some group* of a program's gestures is NP-Hard. We propose the conjecture that finding the minimum number of macros needed to cover *all* of a program's gestures is NP-Hard, although it has not been proven. Formally, we propose that $D4(P, \mu)$ is NP-Complete, where $D4(P, \mu)$ is defined as follows:

$$D4(P, \mu) \;\; (\exists\ S \subseteq I)\ (\exists\ \rho)\ |C(\lambda(S,\ \rho),\ M)| = |I|,\ |M| \leq \mu$$

The optimization form of this problem is then to find:

$$Opt(P) = \min_{n \in N} D4(P, n)$$

We ran into trouble when trying to use our reduction from Subset Sum to prove this conjecture because the Subset Sum problem becomes trivial when we force it to use the entire provided set of integers. We believe that our problem retains its hardness if we were include the entire set of instructions.

In order to complete this proof, then, we would have to think about a reduction from some other NP-complete problem. Rather than starting over from scratch with the proof, we leave this as future work. For the part of our research that involves

finding a good algorithm for field reordering, we operate under the assumption that this conjecture is true and there is no polynomial optimization algorithm.

Chapter 3

Greedy Heuristic for Field Reordering

Given that the problem of finding an optimal alignment for all of the fields in a Java program is NP-Complete (see Chapter 2), we next proceed to find an algorithm that finds a "good" alignment in polynomial time. The result is a greedy heuristic discussed in this chapter.

3.1 Concepts

In order to come up with an algorithm, we created a number of different simple programs and looked at their optimal field alignments in an effort to spot any patterns. The first property that we noticed was that since our goal was to force all fields that appeared in indirection chains to have the same indices, we could never make things worse by moving those fields that are not found in gestures to the end of the field alignment in any arbitrary order. Since our macros would not have to cover these fields, their indices are irrelevant. In addition, even without doing any sort of intelligent ordering it makes more sense for all relevant fields to be at the tops of their respective class alignments, since this increases the likelihood of their having indices in common.

Class Foo	
Foo	a
Foo	b
int	c
double	d

Figure 3.1: Example Field Alignment

p.a.c
q.a.d
r.b.a
s.b.d
t.a.a

Figure 3.2: Example Instruction List (p, q, r, s, and t are instances of Foo objects)

3.2 The Algorithm

The algorithm itself is fairly straightforward. Assume we have all the classes used by a program along with a list of their fields. For example, Figure 3.1 shows the requested information for a sample class `Foo`.

Assume we also have a list of the indirection instructions issued by the program. Figure 3.2 shows a sample instruction list for class `Foo` as defined in Figure 3.1. In this example, the variables p, q, r, s, and t are all previously-instantiated instances of Foo objects.

We next make a chart of all fields in all classes, and for each field sum the number of times that field appears as the first field in a chain, the second field in a chain, and so on. For example, Figure 3.3 shows the properly-constructed chart for Foo defined in Figure 3.1 and the instructions stated in Figure 3.2.

We next use this chart to reorder the fields of each class using the following rules:

- All fields that are not referenced in any chain go at the end of the class, in any order.
- Of the fields that are found in a chain, all reference fields (i.e. those that have a non-primitive type) come before primitive fields.

Field	1st	2nd
a	3	2
b	2	0
c	0	1
d	0	2

Figure 3.3: Example Frequency Chart

Class Foo	
Foo	a
Foo	b
double	d
int	c

Figure 3.4: Example Field Alignment after Algorithm Execution

- Reference fields are ordered first based on the first column of the frequency chart and break ties using the second column. For programs with longer chains, ties in both columns would be broken using the third column.
- Primitive fields are ordered in the same manner once all reference fields have been ordered.

So in the previous example, class `Foo` would have the alignment shown in Figure 3.4. Notice that because `c` and `d` are both primitives and both had a first-column frequency of zero, the tiebreaker was their frequency as the second field in a chain.

Once fields are reordered, the class files must be rewritten using the new ordering. There are a number of open-source tools available that allow the editing of Java class files, including Jclasslib [2].

3.3 Counterexample

Ordering fields in this manner gets very close to the optimal alignment for most programs (as we show in Section greedyseekr), and it does so in a much shorter time than an exhaustive ordering scheme that checks every possible alignment. It does not, however, result in an optimal solution. The following example describes a situation

Class Foo		Class Baz		Class Top		Class Bot		Class Bar	
Top	a	Bot	d	int	f	int	k	Bot	m
Top	c	Bot	e	int	g	int	l		

Figure 3.5: Counterexample Field Alignment

foo1.a.f
foo2.a.g
foo3.c.f
baz1.d.f
baz2.d.k
baz3.e.l
bar1.m.k

Figure 3.6: Counterexample Instruction List

where the algorithm's alignment is very close to–but still greater than–the optimal solution.

Consider the field alignment shown in Figure 3.5 and the list of instructions in Figure 3.6, where the variable bar1 is an instance of a Bar object, baz1 through baz3 are instances of Baz objects, and foo1 through foo3 are instances of Foo objects. Without any reordering, the following four macros cover all instructions: <1,1><1,2><2,1><2,2>.

Using the greedy reordering strategy, we use the instruction list of Figure 3.6 to build the chart shown in Figure 3.7. If we reorder the fields using this chart, we keep all class alignments the same, which leaves us with the same four macros. However, if we switch the positions of `k` and `l` in class `Bot`, we need only the following three macros: <1,1><1,2><2,1>. The greedy algorithm in this case did not arrive at an optimal solution, since by inspection we found a solution better than the greedy solution.

In Sections 7.9 and 7.10 we detail experiments that we did on two implementations of the greedy algorithm to profile its performance. Because there is no efficient optimal algorithm, as we proved in Chapter 2, we are more concerned with the overall performance and speed of the greedy algorithm than its ability to find the optimal solution in every case.

Field	**1st**	**2nd**
a	2	0
c	1	0
d	2	0
e	1	0
f	0	2
g	0	1
k	0	2
l	0	1
m	1	0

Figure 3.7: Counterexample Frequency Chart

Chapter 4

Scavenge Design and Implementation

To implement any ordering scheme, we must first find all gestures in a Java program. The first program that we wrote to find gestures and to test the optimization algorithm outlined in Chapter 3 was named Scavenge. It was a very simplistic implementation that only located double indirections containing two consecutive `getfield` opcodes. In this chapter we discuss the design methodology and features of this program. A more robust version called Seekr that finds all memory-accessing gestures of any length is discussed in Chapter 5.

4.1 Design

Because we were aiming for only a simple way to test the greedy algorithm and count macros, we concentrated on making the code easy to understand and making it work on arbitrary Java programs. To accomplish the former, we commented our code extensively and used Javadoc to create an **Application Programmer Interface** (API), which can be found at `http://deuce.doc.wustl.edu/doc/RandD/ITR/Gestures/Documentation/index.html`. As for the latter, we had to jump through a few hoops to handle the intricacies of the user's possibly-customized environment.

4.1.1 Path Parsing

A major hurdle that we had to cross when considering how to write Scavenge was that we needed to access files in the Java libraries like `java.lang` and `java.util`. These

files can be located anywhere on a user's system as long as the location is disclosed in the CLASSPATH environment variable, as the Java classloader is instructed to scan the colon-separated list of paths denoted by the CLASSPATH variable when searching for `.class` files to load. An example of a valid CLASSPATH is:

```
.:..:/project/cytron/crh2/jdk1.1.8/build/classes/
```

This tells the Classloader to look for `.class` files first in the current directory, then the parent directory, and if it does not find the file in either of these places the specified directory in my user space which contains the java libraries.

The Scavenge executable is a UNIX shell script that first copies the contents of the CLASSPATH variable into a file and then calls the pathParser perl script. pathParser parses each colon-separated path into a temporary file called **paths** that is deleted when the program exits. Therefore, using the example CLASSPATH above, the temporary file would appear as follows:

```
.
..
/project/cytron/crh2/jdk1.1.8/build/classes/
```

Now the information from the CLASSPATH is in an easily-readable format for the Scavenge program so it can recursively search for .class files.

4.1.2 The Reorder Package

The Reorder package is the foundation of the program. It contains the three necessary classes for program execution: IndrFindr, Order, and MacroScavenger.

IndrFindr

The Scavenge program creates an IndrFindr object for each .class file instantiated or accessed by a target program. This object keeps track of any fields in its associated class that have been referenced. It does this by means of a private vector of TableEntry objects, one TableEntry for each referenced field.

While searching the target program for referenced fields, IndrFindr recursively searches each newly discovered Classfile so that all .class files that could be accessed or created during the execution of the target program will be represented by IndrFindr objects. The group of objects is encapsulated in a static vector inside IndrFindr.

Once the recursion has finished, the vector contains IndrFindr objects for every class in the target program as well as all libraries that it uses. Each of these objects also contains statistics (for each of its fields) that are needed for the greedy algorithm, specifically the number of times each field is accessed first and second in an indirection chain.

Order

The reorder method in the Order class actually does the grunt work of the algorithm. It accepts an IndrFindr object and reorders the associated class based on the statistics provided by the IndrFindr object. The reordering is done using the Jclasslib [2] library, which provides an API for opening and manipulating `.class` files. With Jclasslib we could gain access to a class's fields as an array of objects, manipulate this array to order the fields appropriately, and finally rewrite the `.class` file using the new array.

Scavenge iterates through every IndrFindr object in the static vector and calls reorder on each one. Upon completion every class and library used by the target program is reordered based on the statistics accumulated by IndrFindr and the rules presented by the greedy heuristic.

MacroScavenger

For comparison purposes, the reorder package also contains the MacroScavenger class, which computes the number of necessary macros given the current field alignment. When this option is enabled, MacroScavenger runs before any reordering takes place to find the initial number of macros required. After Scavenge creates the IndrFindrs and orders the fields appropriately using Order, the MacroScavenger is run again in order to see how many macros were optimized away.

The structure of this program is very similar to IndrFindr in that it also performs a recursive-descent search of all .class files referenced by a program. However, instead of compiling usage stats, MacroScavenger searches for double indirections and then uses the constant pool to look up the index of each field in the chain. The indexes are then combined together to form a macro of type <`x`, `y`> where `x` and `y` are both integers that correspond to the index of each field. Macros are stored in a static vector inside MacroScavenger, so if the program encounters a unique macro that has not already been stored in the vector, it can add the new macro.

Upon completion, the static vector contains the set of macros needed to cover all double indirections in the target program; these can be displayed elegantly using the displayMacros method.

We use the Scavenge program for a preliminary test of the performance of the greedy algorithm, and the results of this test are found in Chapter 7.

4.2 Usage

Scavenge is checked into the doc-repository in the Scavenge directory. It has the following command-line options:

```
Usage: Scavenge <filename> [-vtime -vstats -vfind -vorder -vtrace
                            -statsonly -help]

    -vtime     Display execution time
    -vstats    Run and display results for MacroScavenger
    -vfind     Display trace information for IndrFindr
    -vorder    Display trace information for Order
    -vtrace    Enable all four previous options
    -statsonly Run MacroScavenger only
    -help      Display usage information
```

The only required command line option is the name of the `.class` file that is the "start" of your program. It is important to run Scavenge on the runnable class or the program will not accurately find all macros.

When running the program on a benchmark or other program that uses its own packages, the top-level directory that contains the packages must be in the CLASSPATH environment variable or the program will not work. To be safe, it's a good idea to put any directory that includes classes you use in your program in the CLASSPATH. Make sure that all CLASSPATH entries contain the full directory structure and not just /directoryName.

For example, to run Scavenge on a program called `/project/cytron/crh2/MyProgram.class` that performs the reordering without displaying the number of macros before and after (as would be done in a real-world situation) while displaying the execution time, use the following command:

```
Scavenge /project/cytron/crh2/MyProgram -vtime
```

The target program must be specified before any command-line options or the program will crash elegantly and display the usage information.

Chapter 5

The Seekr Program

5.1 Design Considerations

5.1.1 Program Motivation

The motivation for writing the Seekr program was to encapsulate a Java program into the format described by our NP-completeness algorithm. Specifically, it generates the indirection-chain instruction list, the type table, and the current field alignment table. We then use this encapsulation to count the number and length of indirection instructions and test the performance of different reordering algorithms.

5.1.2 Motivation for Using Clazzer

In Chapter 4, we discussed the Scavenge program which counts the number of macros needed by a Java program, reorders fields based on our greedy algorithm, and then counts the macros again. However, this program finds and counts only double indirections and only those instructions with two consecutive getfields.

Scavenge was built with Jclasslib [2], but by using Clazzer [9] we take advantage of the data flow modelling abilities to find indirection chains that are split by other bytecode instructions, specifically chains ending with a `putfield`. For most compilers, `putfield` chains are written in the following way: `getfield getfield aload putfield`. This is a triple indirection chain, but Scavenge counts only two `Pgetfields` as part of a chain. With Clazzer, we can construct an instructionGraph that shows the "parameters" of each instruction. This way, the above chain becomes `putfield(aload, getfield(getfield))`.

The data flow method also picks up on multiple indirections that span other instructions. For instance, the following Java instruction sequence would contain a double indirection under one of the possible execution paths of the program:

```
boolean bool = false;
Bar b = new Bar();
Foo f = b.fooInstance;
int result;
if (bool)
  result = f.integer;
else
  result = 3;
```

On one of two possible data flow paths, `int result` is assigned to b.f.integer, which is a double indirection, while the other path is not a gesture at all. Neither assignment opcode comes right after the last getfield. To find this type of indirection, data flow analysis is necessary.

Clazzer gives us the ability to represent a program as a graph whose vertices are connected based on possible dataflow paths. Depending on what kind of information we want, these vertices contain different data about the program. In our case, we use StackMonitorVertices to get information about the instruction stack.

StackMonitorVertices resolve stack operations between `getfields` and track what value is being supplied to a `getfield` rather than just what instruction is before it in the bytecode. They also display the instruction chains in a Lisp-type format. A double indirection of `b.f.integer` is shown as `getfield integer (getfield f(new Bar))`. This makes it very easy to parse the nodes recursively to find chain lengths. Figure 5.1 shows two possible `getfield` chains, one with just one data flow path and one with two.

The Seekr program combines this idea with our original MacroScavenger program. Seekr keeps the two most recently encountered vertices in local variables and goes through the StackMonitorVertices until it comes to either a `getfield` or a `putfield` followed by a non-memory-accessing opcode. When it does, it recursively checks the parameter lists of the getfield or putfield vertex and measures the longest chain along any path to that vertex. With this strategy, all of the types referenced by a program will be scanned, and the longest `getfield` chain will be reported.

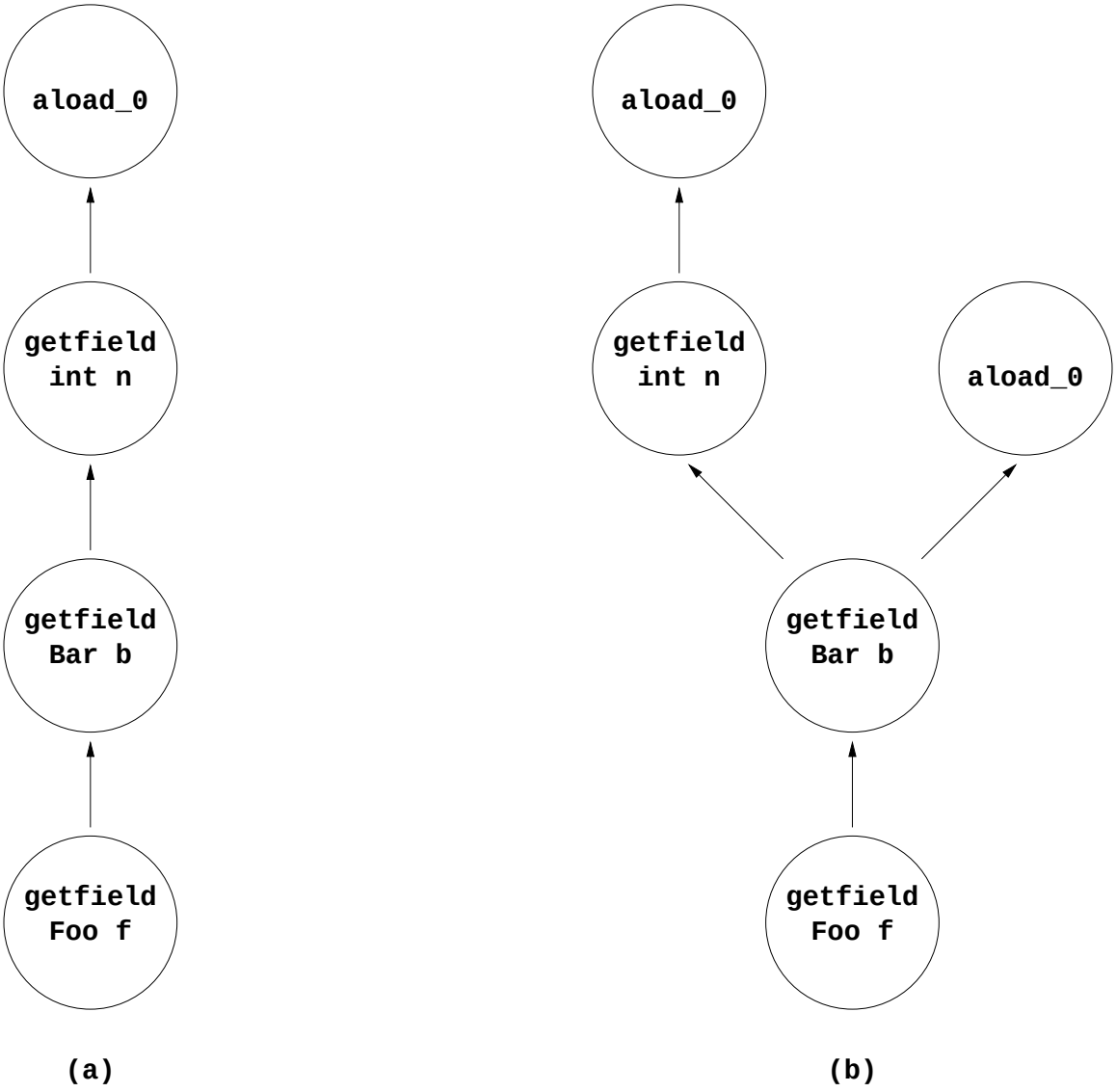

Figure 5.1: Two Getfield Chains: (a) one possible data flow path (b) two possible paths

5.2 Implementation Considerations

5.2.1 Building the Type Table

To build the Type Table, we use a recursive-descent approach and Clazzer along with another utility we wrote called FileFinder. FileFinder uses Java's exception handling capability to find class files in the current directory or the user's CLASSPATH and returns the resulting filename including complete path and extension information. The TypeTable utility begins with a class file specified by the user, reads in its instructions using Clazzer, and finds those instructions that either call methods or declare variables of a different type.

It then recursively finds the class files for these types and repeats the process until it has visited all types referenced by a program. Each class that is visited is appended to the end of a data structure that holds all classes. Once TypeTable is done running, that structure contains the full path location and name of every class file encountered in the specified program. Dynamically, this same task could be accomplished using reflection.

5.2.2 Building the Instruction Table

With Clazzer we can look at the instructions of each class in the Type Table as an instructionGraph. Because related nodes of an instructionGraph contain a reference to the nodes that placed values they use on the stack (for example `getfield(aload_0)`) we only care about nodes that are the end of a chain. From these we can get all information about the rest of the chain just by looking at the interior references. Therefore, we iterate through the instructionGraph and keep references to both the current and last vertices that we encounter. When the current vertex is not a FieldrefInstruction (a `getfield` or `putfield` or a `getstatic` or `putstatic`) while the last vertex *is* one, then the program has encountered the end of a chain.

The program then recursively traverses the graph to find the length of the chain and to create a linked list that encapsulates the instruction as a list of type-name pairs for each field referenced in the chain. Finally it adds this linked list to an ArrayList that stores all instructions, and when the program runs to completion over all classes,this ArrayList is the Instruction Table.

5.2.3 Counting Macros

Given the Type Table and Instruction Table information, it is easy to count the number of macros that are required for an alignment of fields for the types. The countMacro function takes in a HashMap that encapsulates an alignment of fields for all the classes in a program and counts how many macros are needed to cover the instructions in the Instruction Table. Because all of the necessary information is pre-calculated and stored in existing data structures, it can perform this count at an extremely fast rate. In particular, the instruction table does not change, so when checking a large number of different field alignments we can do so without creating any new structures.

5.2.4 The Strategy Pattern

To test the greedy algorithm against other algorithms that reorder fields, we implemented the Strategy pattern [4]. Each algorithm is a class that extends Strategy and has a reorder method that is called to reorder the fields. It is this easy to run tests on each algorithm, and a program could dynamically choose which reordering algorithm to use at runtime. For instance, if the program has a sufficient number of combinations of types to make the exhaustive reordering algorithm take prohibitively long, the program could create and run an instance of the greedy algorithm instead.

5.2.5 Different Strategies

We implemented the following strategy modules for the Seekr program. A more thorough discussion of their implementation and results can be found in Chapter 7.

Random

One strategy for ordering the fields of each class is to generate an order randomly. This random order has no guarantees regarding proximity to the optimal (minimum) number of macros, but in cases where the size of the program is large enough to make exhaustive ordering prohibitive and the number of different orientations that result in a macro count less than or equal to the result from greedy ordering is large, a random strategy may be beneficial. As shown in Chapter 7, it is by far the fastest of the strategies, although for many benchmarks it usually results in a large number of macros compared to greedy.

Exhaustive

Another strategy is to try every ordering and count the number of macros for each one, saving the ordering that requires the fewest macros. This strategy guarantees the minimum number of macros, but makes no guarantees about time. In fact, as the number of classes in a program and the number of fields in each class that are actually referenced by instructions grows, the time to check all combinations grows exponentially. In some cases, the time is between 3 and 5 minutes, but in other cases the time is in the thousands of millennia, which is obviously unacceptable.

Greedy

The greedy algorithm described in Chatper 3 combines speed with effectiveness. In practice it gets very close to optimal in all benchmark trials, and it does so in a very small amount of time (comparable to Random). In fact, most of the time spent in the greedy algorithm is for building the type and instruction tables, since these are the parts of the program that use intensive file I/O. The greedy algorithm does not produce optimal results, however, and in worst-case can be somewhat far away from optimal.

5.3 Usage

The Seekr program and the required TypeTable and Finder utilities are all checked in to the doc-repository under directories of the same name. All three programs are needed to run Seekr, and the locations of the Finder and TypeTable directories must be put into the CLASSPATH environment variable.

Seekr can be run from the Seekr/build directory using the following command-line options:

```
Usage: Seekr [-vtrace -trace -tally -nopf -help -usage] <filename>

  -vtrace  Display verbose trace information
  -itrace  Display instruction trace information
  -tally   Generate chain length statistics for MacroSimulator input
  -nopf    Run the algorithm without considering putfields as
           memory-accessing opcodes
  -help    Display this message
```

```
-usage   Display this message
```

The program will write the macros before and after reordering to files called premacros.out and postmacros.out respectively. The program determines which reordering strategy to use from the second parameter passed into the ClazFindr constructor, chosen from the following:

```
ClazFindr.NONE        No reordering
ClazFindr.RANDOM      Use the random strategy
ClazFindr.GREEDY      Use the greedy algorithm
ClazFindr.EXHAUSTIVE  Check every possible combination and use the
                      minimum
```

We used Seekr to run a number of experiments on the Java benchmarks, including a test of all the different reordering strategies as well as a profile of the benchmarks themselves. These tests are discussed in Chapter 7.

Chapter 6

The MacroSimulator Program

To get a better picture of the time saved from using our macro strategy for chains of `getfield` and `putfield` instructions, we developed a graphical package that could simulate the execution of a Java program.

6.1 Input: The .sprob and .sin files

We designed our input file so that it could be populated with data taken from an actual Java program or generated randomly for easy testing. An `.sprob file` (Simulation PROBability) contains a probabilistic representation of a finite state machine. This state machine represented a Java program in that it could be used to build a series of opcodes that are either `getfield`s or non-indirection instructions. As a chain grows, the probability that the next instruction will be a `getfield` goes down. We then wrote a Perl script called sprob2sin that would translate this representation into a more structured program-like representation using the given probabilities.

After studying a few of the output files, however, we realized that it would be much simpler to treat the chains as "blocks" rather than treat each instruction atomically. We made this decision because a program could have length-1 chains and length-3 chains but no length-2 chains, and in our probabilistic model it resulted in more longer chains than would be expected. Therefore we altered our protocol so that an `.sprob` file is specified as follows:

- Total number of "blocks" (single instructions plus chains)
- Number of single instructions

- Number of chains of exactly length 1 ending with the `getfield` opcode
- Number of chains of exactly length 2 ending with the `getfield` opcode
- Number of chains of exactly length 3 ending with the `getfield` opcode
- Number of chains of length 4 or greater ending with the `getfield` opcode
- Number of chains of exactly length 1 ending with the `putfield` opcode
- Number of chains of exactly length 2 ending with the `putfield` opcode
- Number of chains of exactly length 3 ending with the `putfield` opcode
- Number of chains of length 4 or greater ending with the `putfield` opcode

Our Perl script uses the probabilities (each individual tally divided by the total number of blocks) to generate a sequence of tokens equal to the total number of blocks. In a large enough sample size, the ratios of the randomly-generated program should tend to be about the same as the ratios in the original source program.

Each token in a `.sin` file (Simulation INstruction) is an integer that represents a chain length. For the average program, most tokens are 0's, which represent a standard Java opcode that does not reference memory. A standalone `getfield` or `putfield` that is not part of a chain is denoted by a 1. A chain of `getfield`s that ends in either a `getfield` or `putfield` is denoted by its length (including the closing `putfield` if there is one present).

6.2 Using Seekr to Generate Simulator Input

To generate realistic `.sprob` files that would give us meaningful data, we added functionality to the Seekr program that previously just searched for gestures. We added 9 tally variables that keep track of different lengths/types of gestures. There is a different variable for each length gesture from 0 to 4 or longer, then separate variables for 1 through 4+ length gestures that end in putfields, since in the simulator the time to execute a `putfield` and a `getfield` are different. When the tally option is enabled, the program dumps the resulting numbers for each different length and type of gesture to a file called `log.sprob` along with the total number of instructions.

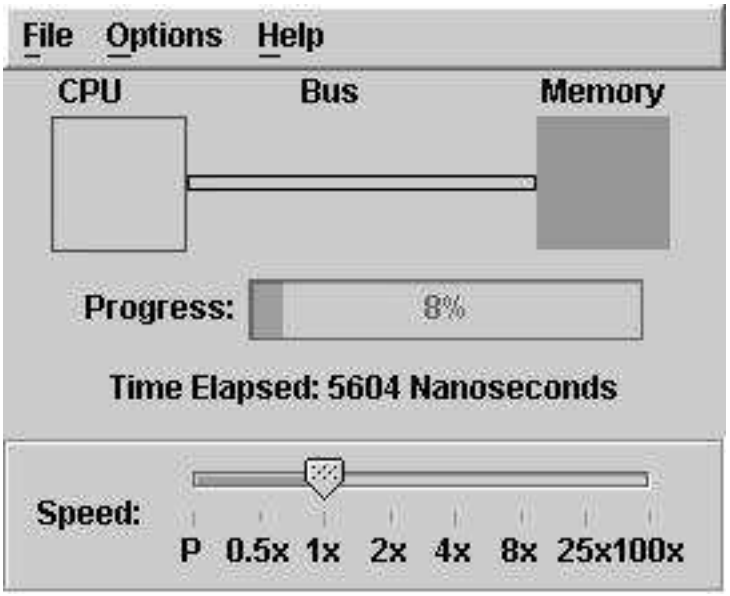

Figure 6.1: The Simulator Main Screen

6.3 The ScavengeModel Class

This class is the guts of the simulator and is responsible for the actual simulation. It takes in a `.sin` file and "executes" the program by calculating the amount of time it would take to execute the instruction or chain based on provided time constants. The program specifically takes into account time spent charging and using the memory bus and time spent fetching data on main memory. All of these variables can be set on the command line, and the program can also run a batch of executions in order to get an average.

Upon completion, the program displays the total time to execute the program. If MacroSimulator is running in batch mode, it will output the average time for a single run (average to provide a general sense of the execution time for all runs).

6.4 The Visualizer

An optional plugin for the ScavengeModel is a Java Visualizer that allows the user to open files and tweak variables using a traditional, menu-driven GUI as well as monitor execution time and progress with a graphical representation. The representation includes a display of where the process is currently operating (CPU, bus, or main

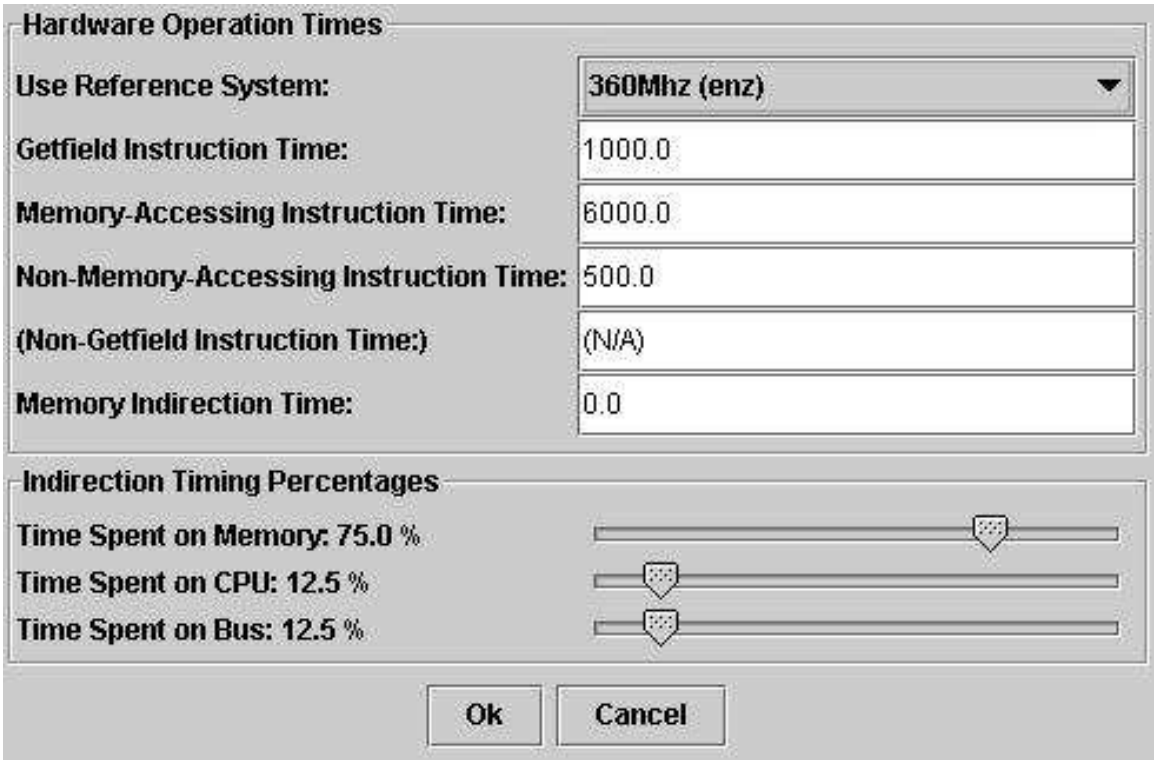

Figure 6.2: The Hardware Options Screen

memory) as well as a way for the user to adjust simulation speed (which does not affect the timing numbers, only the speed in which instructions are displayed). Figure 6.1 shows a screen shot of the visualizer in action. The sliding bar at the bottom controls simulation speed relative to "real-time".

The Visualizer also has an easy menu system for inputting parameters such as hardware speed and whether or not to use macros. The hardware menu, shown in Figure 6.2 has built-in presets for a number of different processor configurations with actual timing numbers taken from all of them. Using one of the presets, the user can simulate the macro system on a number of different machines and measure the impact of hardware speed on the effectiveness of the system.

The software options screen (Figure 6.3) lets the user set the percentage of non-gesture instructions that access memory and specify whether or not to use a simulated PIM system with macros to execute indirection chains.

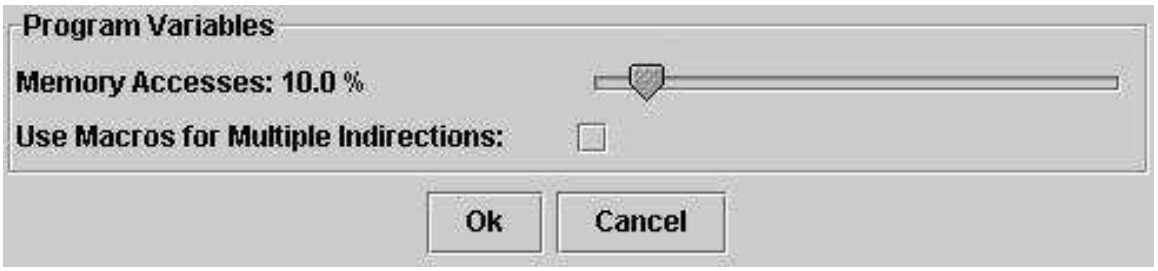

Figure 6.3: The Software Options Screen

6.5 Usage

The MacroSimulator program has the following command line options; by default, the visualizer is enabled.

```
java MacroSimulator [-f filename] [-m] [-q] [-h]

   -m       Multiple Indirections handled entirely by memory
            (Otherwise on CPU).
   -q       Quiet Mode (Visualizer disabled)
   -h       Help
   -b num   Batch mode, where num is the number of times to run the
            program.  Will output the average execution time over
            all runs.
   -f file  Load a file on startup, where file is the filename.
```

The -m switch is the command to use a simulated PIM to handle indirection chains. Running the same program with and without this switch enabled will result in a good approximation of times for the same program to run with and without macros and provides a way to quantify the speedup.

When the program is run with the Visualizer enabled, there are a number of keyboard shortcuts for common operations. Along with the ones shown in the menus, the following shortcuts manipulate the simulation speed on the fly:

```
P       Toggles between paused and running simulation
[up]    Increase simulation speed
[right] Increase simulation speed
```

```
[down]  Decrease simulation speed
[left]  Decrease simulation speed
```

Our experiments using the Simulator are presented in Chapter 7.

Chapter 7

Experiments

Over the past two years, we have run a number of different experiments using the software that we implemented. At first, the goal of our tests was to determine the length of macros that are found in typical Java programs. Once we determined that indirection chains did indeed exist, we wanted to know how many there were and how to find them efficiently. The ultimate goal, though, was to test the greedy reordering scheme outlined in Chapter 3 and measure its performance against other reordering schemes. In this Section we detail all of the experiments we did and discuss their results.

7.1 The Java Benchmarks

For most of the tests discussed here, we used the SPECjvm98 [14] collection of Java benchmarks, which consists of the following programs:

_200_check: A program that tests features of the JVM including array indexing, inheritance, and loops.

_201_compress: Modified Lempel-Ziv method that finds substrings and replaces them with variable-sized code on the fly.

_202_jess: The Java Expert System Shell that solves a set of puzzles.

_205_raytrace: A ray tracer that works on a picture of a dinosaur.

_209_db: Performs database functions on a database residing in memory.

_213_javac: The Java compiler that comes packaged with JDK 1.0.2.

_222_mpegaudio: An audio decompression utility.

_227_mtrt: A multithreaded variant of **raytrace**.

_228_jack: A Java parser generator.

7.2 Finding Indirection Chains with Javap

The first experiment we did was to find gestures using the `javap` [10] utility that comes packaged with most Java installations. With the -c command-line switch we were able to get a complete listing of the opcodes for each method of a Java `.class` file. In this case we use the word `gesture` to define any group of consecutive `getfield` opcodes of length greater than 1 that is surrounded by non-`getfield` opcodes. We wrote a Perl script [3] to count the number and length of gestures in a single directory of `.class` files. The script itself can be downloaded from the doc repository, where it is located in the LucasFox/javap directory. To run it on a directory, simply run the lucasTest script. The default benchmark location is /project/cytron/cytron/jvm98/, but this line can be changed in the script to point to the directory to be scanned.

We chose to analyze directories as units because the set of Java benchmarks that we used has each separate program in a single directory, and the Java packages like `Math` and `IO` also have all compiled classes in a source directory. The results of this experiment can be found in Section 7.11.1.

7.3 Benchmark Statistics

We used the Seekr program outlined in Chapter 5 to gather some general data about the Java benchmarks and get a more accurate count of length-2 and length-3 chains. These counts now include `putfield` instructions in the chains thanks to the data flow analysis capabilities of Clazzer. The information from this test helped to make connections between program size and number of combinations and processing time as well as provide a gauge for whether there are enough indirection chains in an average Java program to make our macro idea worthwhile.

By adding some basic counters to Seekr, we were able to tally the number of classes in the type table and the number of instructions in the instruction table. This way, we could get some idea of the size of each program in terms of number of classes and number of indirection instructions. We also did a simple calculation to compute

the number of combinations for the exhaustive search. We multiply the number of referenced fields in each class together to find the number of combinations, since only referenced fields need to be permuted and considered. The results of running this verbose version of Seekr on the Java benchmarks can be found in Section 7.11.1

7.4 Seekr Timing Analysis

While Seekr is a much more robust and accurate program than Scavenge, it also takes longer to run when performing the greedy reordering. This is not a big concern, since the utility would only need to be run once on a Java program after compilation, but we still wanted to get a better idea of which stage of execution caused the slowdown.

We used the Java System package's currentTimeMillis function to get the elapsed execution time at different points of the program in order to put together a timing analysis for each benchmark. The analysis calculates the percentage of total runtime dedicated to type table construction, instruction table construction, initial macro count, reordering, and final macro count. The results of the Seekr timing tests on all Java benchmarks are in Section 7.11.2

7.5 Indirection Chain Execution Timing

The premise of this thesis is that we would gain some benefit from replacing longer indirection chains with, from the CPU's point of view, single calls to main memory. But what if subsequent memory accesses are already much faster than the initial access time in a chain? In order to verify the hypothesis that all memory accesses take approximately the same amount of time, we measured the execution time in a small Java program of indirection chains with lengths between one and five using a high-resolution timing package.

One interesting aspect of the experiment we encountered was that if we accessed the same fields twice in a test, the second access would take a much shorter time because the field contents would be held in system cache. Therefore, we tested each chain length using entirely different objects and fields to ensure that cache would not impact the times. We also measured the overhead of starting and stopping the high-resolution timer and subtracted that time out of the final measurements. Thus, the times that we measured should be accurate, taking into account only the execution of the indirection chains themselves.

After measuring the times for each length and plotting them on a graph, we used linear regression to fit a line through the points. The slope of this line represents approximately the time penalty for each additional `getfield` instruction in a chain. The graph and analysis of the results can be found in Section 7.11.2

7.6 Measuring Idea Feasibility with MacroSimulator

Now that we knew the approximate execution times for indirections, we used these numbers in the MacroSimulator to get a decent approximation for the speed gain we could expect from our idea. We ran the benchmarks through Seekr to get the required .sprob files then converted them to `.sin` files using sprob2sin. Finally, we ran the `.sin` files through MacroSimulator both with and without using macros to see what kind of numbers we got. We used an average of the execution times of a few of Java's non-memory-accessing opcodes when configuring the simulator to simulate instructions not part of a gesture. The results of these tests are in Section 7.11.2.

7.7 Exhaustive Reordering

Because we wanted to gauge the success of a reordering strategy, we should know the absolute minimum and maximum number of possible macros in a program. The problem of finding an optimal reordering was shown to be NP-complete in Chapter 2, so the only way (assuming $P \neq NP$) to find the maximum and minimum is to check every possible field ordering for every class in a program. Because we designed Seekr to be extensible in terms of reordering strategies, to run this test we implemented a module that performs this exhaustive check. We called this module ExhaustiveStrategy.

There are two stages to my method of reordering: the permute stage and the combine stage. In the permute stage, we generate the possible orderings of fields for each class. In the combine stage, we combine the classes together so that every ordering of every class is tested with every ordering of every other class.

7.7.1 The Permute Stage

At first we tried to store every permutation of fields for each class, but this turned out to be prohibitively costly. One class has 19 fields, yielding $19! = 1.22 \times 10^{17}$ different permutations for just this class. This is easily large enough to cause the process to run out of memory.

However, a simple heuristic that we incorporate into the Greedy strategy but which also holds for any optimal strategy is that fields that are never accessed anywhere in the program can always be put at the end of every permutation for their class. Therefore we only need to permute the subset of fields that are accessed by the program in question and can append the remaining unused fields in any order as long as we keep that order consistent.

By doing this, we cut down the permutation storage substantially, and it enabled us to be able to permute the fields of all classes in the Java benchmarks in a very reasonable (no more than a minute or so) amount of time.

7.7.2 The Combine Stage

Initially we tried to use recursion and a large data structure to store all possible legal combinations of the the class permutations for a program. However, the number of combinations grows at an alarming rate based on the number of permutations there are for each class. For example, we estimated the number of combinations required for the Check benchmark at 6.688×10^{10}. This is also prohibitively large in terms of size and space, and it causes the process to run out of memory during calculation.

Instead we calculated each combination on the fly using an odometer-like system. In this system, we strung together the different classes into a linked list with each permutation of that class acting as a "digit". Combinations are formed by combining the current "digits" of each class, then rotating the rightmost class to the next "digit". Once the rightmost class cycles through all of its permutations, it returns to the starting "digit" and cycles the class immediately to its left by one permutation. This behavior extends to all classes, so when the leftmost class cycles through all of its permutations then we know that all possible combinations of "digits" have been generated.

This approach allowed us to cut down immensely on storage space since we could check the combination immediately and not have to save each one, and it also reduced processing time since we would not be enduring the overhead of recursive calls

or spending the time to construct a large data structure and then iterating through it. We ran the exhaustive search on each Java benchmark, and the results are in Section 7.11.3.

7.8 Random Reordering

A problem with using the Exhaustive search numbers to approximate upper and lower bounds for the benchmarks whose size prohibits us from running to completion is that we are forced to make extrapolations after having checked only a small percentage of the data. To get around this, we implemented a RandomStrategy reordering module for Seekr that chooses a class permutation randomly for each class in the program and measures the number of macros needed to cover this alignment. By running this program many times, we can get a good random sample of field alignments and measure statistics such as mean and standard deviation over each iteration. This technique alleviates the problem introduced by the odometer method, specifically that the "high digits" will never get checked in large programs, but it introduces a randomness factor that removes any guarantees about finding actual minima or maxima.

As with the exhaustive test, we ran Seekr with the RandomStrategy module enabled on all of the Java benchmarks. The results are discussed in Section 7.11.3.

7.9 Greedy Reordering using Scavenge

Now that we had some baselines for comparison, we set out to test our greedy heuristic. The first test of the greedy strategy, which is outlined in Chapter 3, was a simple test using the Scavenge program. Recall that because of limitations in Scavenge – specifically that only length-2 chains are treated – the results of this test would not be totally accurate, but they did give us some idea of whether or not the greedy algorithm was feasible to implement and whether it gave us any sort of improvement over the out-of-box (OOB) field alignments of the benchmark classes. The results of this test can be found in Section 7.11.3.

7.10 Greedy Reordering using Seekr

The most important test of the greedy strategy that we ran was to test it on Seekr. This way we could compare its results to the results of the exhaustive and random strategies in the same framework. Unlike the greedy test with Scavenge, this implementation took into account chains of any length that included both `getfield` and `putfield` opcodes. We implemented a strategy module for Seekr called GreedyStrategy that would perform our heuristic and put it to work on the Java benchmarks. We discuss the results in Section 7.11.3.

7.11 Results

In this section we outline and discuss the results of all the experiments described in the previous sections. The experiments are divided up into three groups based on what we set out to measure in each one. The groups are indirection chain count, timing, and reordering.

7.11.1 Indirection Chain Count Results

Javap

The results of the search for indirection chains using `javap` as outlined in Section indirjavap were not very encouraging. Ideally we would have liked to see programs with long getfield chains, as this would enable us to bypass the greatest number of back-and-forth motions between the CPU and main memory. However, none of the benchmarks nor the Java packages had any chains longer than length 2, and only three of the seventeen directories checked had more than 14 length-2 chains. The complete results are printed in Figure 7.1.

The discouraging results of this test did not cause us to give up, though. For one thing, this script does not detect `putfield` instructions, since in Java byte code a chain ending with the `putfield` operator has a `push` instruction in the middle that puts the value to assign the field onto the stack.

As we discussed in section Section 1.1, the `putfield` instruction is still part of a chain even though it does not come directly after the `getfield` operator. By leaving these operators out, the script finds no length-3 chains, but we expect that a more robust test that included `putfield` opcodes would find some.

Package	Double Indirections	Triple Indirections
_200_check	0	0
_201_compress	0	0
_202_jess	0	0
_205_raytrace	3	0
_209_db	0	0
_213_javac	216	0
_222_mpegaudio	102	0
_227_mtrt	0	0
_228_jack	14	0
_999_checkit	0	0
java.io	4	0
java.lang	0	0
java.math	0	0
java.net	0	0
java.security	0	0
java.text	44	0
java.util	3	0

Figure 7.1: Gesture Counts using Javap -c

Benchmark Statistics

Figure 7.2 shows the results of the more robust benchmark test using data flow analysis discussed in Section 7.3, and clearly we were right in our conjecture that length-3 chains do exist. In fact, in many of the benchmarks triple indirections are more plentiful than double indirections, which is good news for us.

javac is by far the largest benchmark in terms of classes and indirection chains, with jess, jbb, and check challenging for a distant second place. These numbers are much more promising than the ones we obtained from the Javap script. Based on the statistics, we could see some definite performance gains in a number of the benchmarks with the macro implementation.

Incidentally, we used these numbers to calculate how much replacing consecutive memory-accessing opcodes with a single macro-calling opcode would reduce code size. Assuming that `getfield` and `putfield` instructions both are 3 bytes (1 byte for the opcode, 2 bytes for the constant pool index) and that our macro-calling opcode would also be 3 bytes (with the second 2 bytes now being an index into the macro table), we calculated the space savings shown in Figure 7.2. The reductions turned

Benchmark	Classes	Chains		Combinations	Size Reduction
		Double	Triple		
_200_check	65	48	18	66886041600	252 bytes
_201_compress	49	3	13	2304	87 bytes
_202_jess	144	26	32	31850496	270 bytes
_205_raytrace	76	7	16	331776	117 bytes
_209_db	58	3	13	2304	87 bytes
_213_javac	155	183	49	9.82×10^{21}	843 bytes
_227_mtrt	76	7	16	331776	117 bytes
_228_jack	88	18	24	55296	198 bytes
_999_checkit	59	3	16	13824	105 bytes
JBB	91	58	32	5.55×10^{15}	366 bytes

Figure 7.2: Benchmark Statistics

out to be negligible when considering the overall size of the class files comprising the benchmarks, shown in Figure 7.3.

7.11.2 Timing Results

Seekr Timing

The results of the timing tests on Seekr outlined in Section 7.4 can be found in Figure 7.4, with all numbers representing percentage of benchmark execution time. The percentages are graphed in Figure 7.5. The first two columns represent the time to construct the type and instruction tables respectively. The final three columns are the time required to count the macros before reordering, perform the greedy reordering algorithm, and do the post-reorder count.

Clearly the majority of the time is spent constructing the type and instruction tables. Because the type table is built recursively using an imported package, we incur extra overhead in the added method calls, which explains the slowdown here. The instruction table also uses a lot of the time-intensive functionality of the Clazzer package.

One interesting thing to note is that in the larger benchmarks in terms of number of classes and number of gestures, the greedy reordering takes a much larger percentage of time than in the smaller benchmarks. This implies that the addition

Benchmark	Size (bytes)
_200_check	32428
_201_compress	17745
_202_jess	345830
_205_raytrace	48999
_209_db	10064
_213_javac	1926681
_227_mtrt	119982
_228_jack	707
_999_checkit	5245

Figure 7.3: Benchmark Size

of more classes or gestures incurs a greater time penalty in the reordering algorithm than in table construction.

The larger benchmarks do not have a marked increase in percentage of time devoted to macro counts, however. This is an expected and desirable result that means we can check lots of possible alignments in a reasonable amount of time.

Indirection Chain Timing

The resulting graph of the points and the fit line, Figure 7.6, shows that our hypothesis from Section 7.5 was correct and that the relationship between chain length and execution time is close to linear. The slope of the line is 8342.8ns per `getfield` instruction, so we should expect to save about this amount of time for double indirections and about twice that for triple indirections. Of course, we only get this savings in a system without cache.

In order to illustrate the incredible effect of cache, we graphed the execution time for single, double, and triple indirections both with and without cache. The results are in Figure 7.7. Clearly our work would not be much help on a system with cache.

MacroSimulator Results

We ran the simulator both with and without macros on all of the benchmarks using timing information taken from a Sparc system. The results are shown in Figure 7.8.

Benchmark	Type Table Construction	Instr. Table Construction	Initial Count	Reorder	Final Count
_200_check	48.48	37.17	0.16	14.19	0.01
_201_compress	57.6	38.48	0.08	3.85	0
_202_jess	52	40.89	0.16	6.95	0
_205_raytrace	53.58	41.1	0.08	5.24	0
_209_db	55.27	41.5	0.08	3.15	0
_213_javac	36.16	49.14	0.6	14.09	0
_227_mtrt	52.74	41.93	0.11	5.22	0
_228_jack	50.39	42.04	0.09	7.48	0
_999_checkit	53.82	41.02	0.09	5.06	0
JBB	47.38	37.44	0.26	14.92	0

Figure 7.4: Seekr Timing Profile

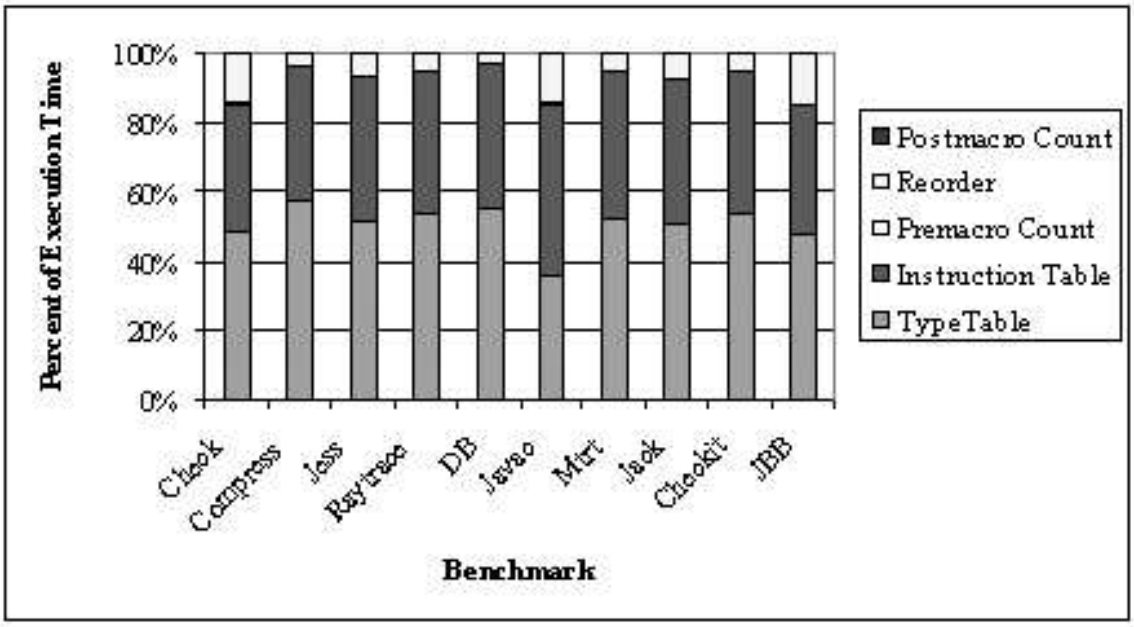

Figure 7.5: Timing Analysis of Benchmarks

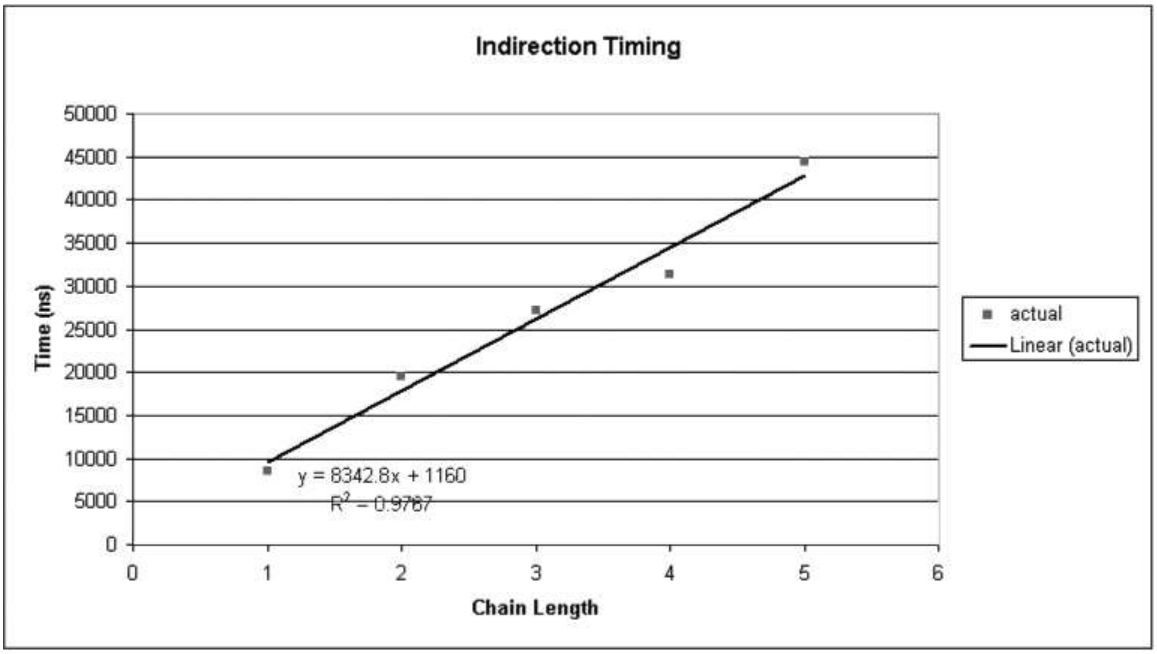

Figure 7.6: Execution Time of an Indirection Chain vs. Chain Length

There is a good savings using macros for all of the benchmarks, especially jess and javac. The graph in Figure 7.9 shows these numbers as well.

7.11.3 Field Reordering Results

Exhaustive Reordering

Using the system described in Section 7.7, we were able to get upper and lower bounds on the number of macros for most of the Java benchmarks. Despite the optimizations to our program, some of the benchmarks still took too long to run. While the optimizations significantly reduced the number of combinations in most cases, a significant reduction from a very large number is still a very large number.

For the benchmarks that would have taken prohibitively long to complete, we had the exhaustive search module calculate the total number of field combinations there are for each benchmark and how long it would take to finish checking all of them. We found that, on average, the program can examine about 100,000 combinations per minute on the machine we ran the tests on, and we used that number to calculate time to finish. Figure 7.10 shows the calculated bounds and times for each benchmark.

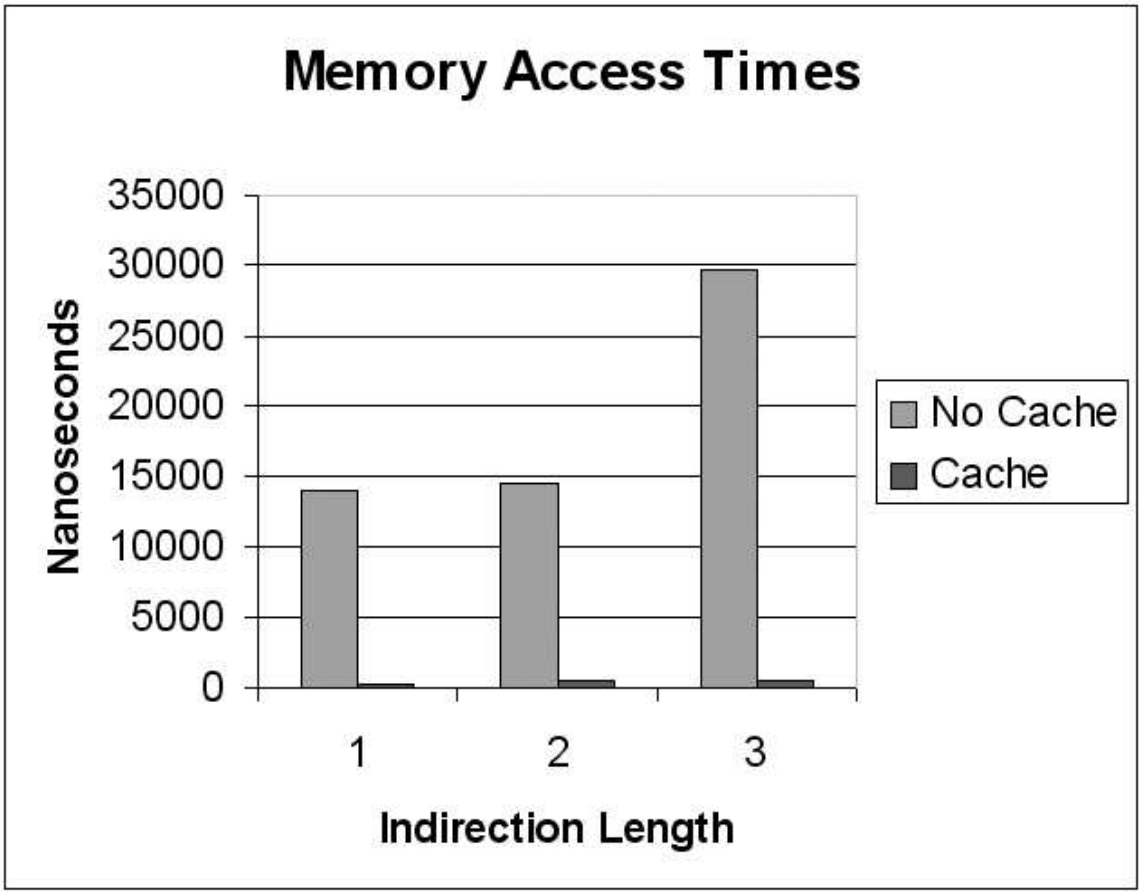

Figure 7.7: Indirection Chain Timing With and Without Cache

Benchmark	Time (ns)		Percent Gain
	Without Macros	With Macros	
_200_check	29510584	28066152	4.89%
_201_compress	15043107	14570036	3.14%
_202_jess	30694756	28813258	6.13%
_205_raytrace	21161973	20168000	4.70%
_209_db	17458063	17034889	2.42%
_213_javac	52677663	49832393	5.40%
_227_mtrt	21347262	20245396	5.16%
_228_jack	28316428	27343575	3.44%
_999_checkit	16260869	15403690	5.27%

Figure 7.8: Time Savings of the Macro Strategy from MacroSimulator

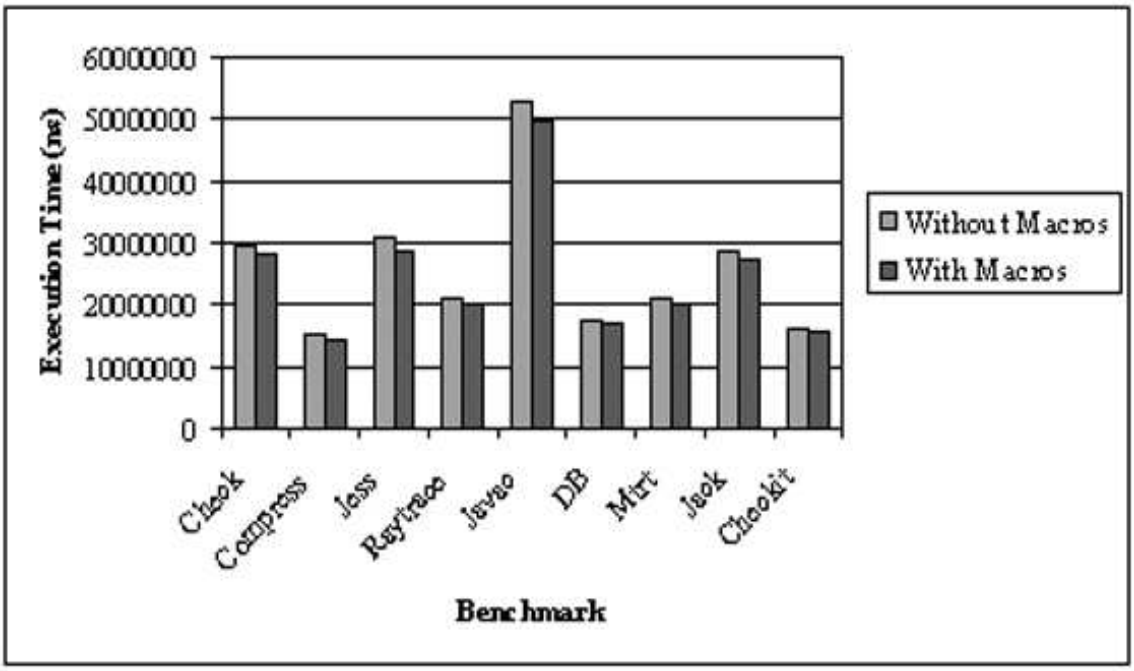

Figure 7.9: Graph of Simulated Execution Time With and Without Macros

Benchmark	Min	Max	Done	Combinations	Completion Time
_200_check	17	24	No	66886041600	1.27 Years
_201_compress	8	9	Yes	2304	1.38 Seconds
_202_jess	10	13	Yes	31850496	5.31 Hours
_205_raytrace	11	14	Yes	331776	3.32 Minutes
_209_db	8	9	Yes	2304	1.38 Seconds
_213_javac	26	34	No	9.82×10^{21}	1.86×10^{8} Millennia
_227_mtrt	11	14	Yes	331776	3.32 Minutes
_228_jack	11	12	Yes	55296	33.18 Seconds
_999_checkit	9	12	Yes	13824	8.29 Seconds
jbb	21	29	No	5.55×10^{15}	105 Millennia

Figure 7.10: Resulting Bounds from Exhaustive Ordering

For JBB, the exhaustive strategy did not run long enough to encounter the actual lower bound (we know this because the Greedy strategy results in 20 macros, as discussed in Section 7.11.3). This is the danger of using this method to measure bounds when the program does not finish. We have no real guarantee that any of the incomplete tests came up with a number even reasonably close to the real minimum for those programs, but without the aid of an extremely fast computer, they are the best results we can get.

Random Reordering

Figure 7.11 shows the results of running the random reordering module on the Java benchmarks. For each benchmark, we ran the program 25 times and measured the mean number of macros and the minimum for that set of trials. From these results we can really see the benefit of making the basic assumption that all referenced fields should be placed before all unused fields in each class. Because the random strategy does not make this assumption when randomly positioning each field, the mean is in all cases higher and in many cases much higher than the maximum value obtained from an exhaustive search.

We would expect that an increased number of trials per benchmark would result in a lower minimum, but the 25 trials already take upwards of 15-20 minutes to run for each benchmark. It is also worth noting that we ran the trials a few times during the development process, and the means were very consistent over all cases. This leads us to believe that even upping the number of trials considerably will not have any effect on the average and that the observed mean is accurate. Clearly a strategy involving random ordering is not desirable for any sort of real-world situation.

Greedy Reordering Using Scavenge

Figure 7.12 shows the results of running Scavenge on each of the benchmarks as outlined in Section 7.9. The time represents the full runtime of the program, which includes an initial macro count, the greedy reordering, and a final macro count to judge the benefit. The program can also be run without the two macro counts; the greedy algorithm runs in exactly the same manner but experimentally we cannot verify the results. If the program were run in a real-world situation, we would not care about the measuring the performance benefit, so we have also included the time it takes to run the algorithm with the macro counting disabled. Since any sort of

Benchmark	Mean (Macros)	Min (Macros)
_200_check	25.92	24
_201_compress	10.76	9
_202_jess	19.48	16
_205_raytrace	16.76	14
_209_db	11.04	10
_213_javac	55.52	49
_227_mtrt	17.76	15
_228_jack	16.72	15
_999_checkit	13.56	12

Figure 7.11: Mean and Minimum from 25 Random Trials for Each Benchmark

reordering program would be a run-once application, a maximum time of around 20 seconds is pretty good.

Greedy Reordering Using Seekr

Once again we ran the strategy described in Section 7.10 and then immediately counted the number of macros needed for program coverage. Armed with this information, we put together all of the results from the different Seekr strategy modules into one table. It is worth nothing that in terms of runtime the random strategy is the fastest but it also is the least consistent. The exhaustive and greedy strategies will return the same results for the same programs every time, but the exhaustive strategy has the potential to take a much longer time to run.

Figure 7.13 shows the number of macros obtained from doing a greedy ordering as compared to the number out of the box, the maximum and minimum from exhaustive, and the minimum from a random search of 25 trials. Of the benchmarks that completed exhaustive reordering, the greedy algorithm came up with the optimal alignment in every case. The numbers for greedy, maximum, and minimum are graphed in Figure 7.14.

For the benchmarks that do not finish the exhaustive search, it is harder to interpret results. In JBB, for instance, the greedy algorithm comes up with 20 macros while exhaustive finds 21, but since exhaustive only checked one one-thousandth of a percent of all possible combinations, we cannot make any statements about greedy compared to this number.

Benchmark	Macros		Time (s)	
	OOB	Post-Greedy	(With Count)	(No Count)
_200_check	13	8	27.593	8.98
_201_compress	2	2	7.441	4.783
_202_jess	6	3	24.196	15.304
_205_raytrace	4	3	11.792	8.266
_209_db	2	2	8.964	5.438
_213_javac	19	9	63.928	19.93
_227_mtrt	21	16	15.466	8.04
_228_jack	3	3	11.303	11.747
_999_checkit	6	2	14.834	8.832

Figure 7.12: Performance Benefit of the Greedy Algorithm Using Scavenge (OOB is Out of Box)

Benchmark	Macros				
	OOB	Maximum	Random	Greedy	Minimum
_200_check	19	24	24	17	17
_201_compress	9	9	9	8	8
_202_jess	17	13	16	10	10
_205_raytrace	13	14	14	11	11
_209_db	8	9	10	8	8
_213_javac	41	34	49	28	26
_227_mtrt	11	14	15	11	11
_228_jack	14	12	15	11	11
_999_checkit	9	12	12	9	9
JBB	30	29		20	21

Figure 7.13: Algorithm Comparison in Seekr

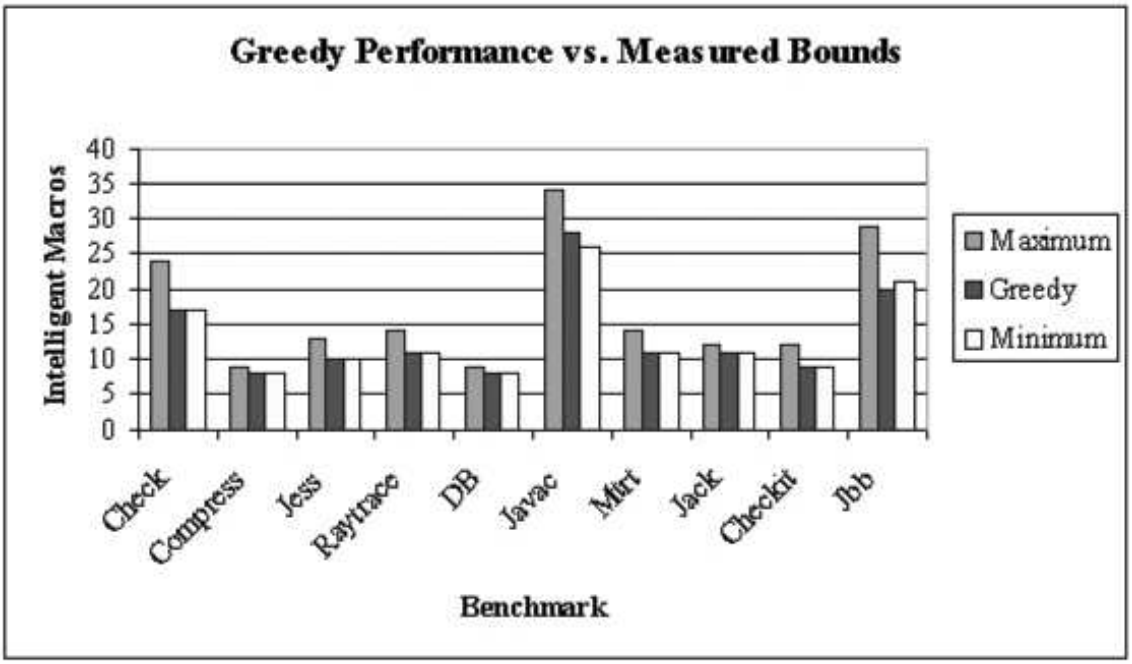

Figure 7.14: Graph of Greedy Performance Against Measured Bounds

Since the runtimes are much shorter for greedy than for exhaustive on most of the benchmarks, and because the greedy strategy gives us a guaranteed result where random does not, we would say that our heuristic performed quite well on whatever program we threw at it.

7.12 Experimental Conclusions

From running all these experiments, we learned that indirection chains do exist, they can be found with javap -c but are found in greater numbers with data flow methods, and reordering the fields in each class based on a simple heuristic reduces the number of macros we need. We also discovered that the execution time for indirection chains is linearly based on the length of the chain as long as none of the addresses to fetch are cached.

Chapter 8

Conclusion

In this document, we designed and outlined a method for describing a series of memory-accesses in Java bytecode. We also described a means for expressing these series opcodes as gestures. At first we had no idea of the quantity or length of gestures in real Java programs, but through experimentation we gathered enough data to show that they are relatively plentiful.

Next we introduced the idea of macros on a PIM device which would do the repeated fetching of a gesture without sending intermediary information back and forth to the CPU. Much of my work was exploring the various ways of ordering the field of Java `.class` files so that all gestures can be translated into the fewest number of macros.

After showing that finding an optimal solution is an NP-complete problem, we outlined a greedy algorithm and measured its effectiveness against a variety of other methods including random ordering and a time-consuming exhaustive search. In doing this, we implemented two programs that could actually reorder the fields of a Java `.class` file. The first was a simplistic program for testing purposes only, but the second is much more robust application that conceivably could be used in real-world situations.

Finally, we authored a piece of customizable simulation software that let us visualize the benefits of the PIM without actually implementing it in hardware. By specifying the timing data for any system, this simulator could be used to measure the performance of our macro strategy on that system.

8.1 Future Work

Based on our research, future work could actually implement the PIM using an FPGA or other programmable hardware device. We successfully replaced a gesture with a homemade custom opcode in a real Java program to test the feasibility, but a program would have to be written to not only replace all gestures but also to interpret the new custom opcode in the JVM. There are a number of free opcodes that Sun left for programmers to define as they see fit, but any attempt at running a program using one of these opcodes on an existing JVM would fail. There are any number of open source Virtual Machines that run under various environments, and any of these could be adapted to handle the new opcodes and communicate with the PIM using assembly or some other low-level machine code.

Another interesting issue is that of cache. Because the time it takes to run indirection chains that access main memory relates linearly with the length of the chain, we could expect a good speedup if we push this processing to some sort of intelligent processor in main memory.

However, indirections chains that accessed cached information did not take longer to execute as chain length got longer. In fact, a long chain accessing only cached data ran even faster than a single access of main memory. Therefore, forcing a program to always use main memory may end up slowing down one that contained a lot of repetitive memory fetch patterns. Future work could be done exploring this phenomenon and perhaps developing some sort of controller that could decide whether to go straight to cache or use our PIM based on which would result in a faster fetch time. Our concept could also be implemented directly in cache.

8.2 Final Thoughts

Ever since its infancy, Java has been both praised for being easy to learn and scorned for being bulky and slow. Now that the language is becoming more mature and the first language of a new generation of programmers, it is more important than ever that we make every effort to keep the front-end simple while making the back-end more efficient.

The idea of a PIM, if made to work in coordination with system cache, could result in a smaller code footprint and faster execution for many programs without

any additional work on the user end and only a small amount of one-time additional overhead at compile time.

References

[1] Matt Curtin. Write once, run anywhere: Why it matters. `java.sun.com/features/1998/01/wora.html`, 1998.

[2] ej-technologies GmbH. Jclasslib 1.1. `www.ej-technologies.com/products/jclasslib/java.html`, 2001.

[3] Lucas M. Fox. Memory-Accessing Optimization Via Gestures. Master's thesis, Washington University in St. Louis, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, San Francisco, CA, 1979.

[6] R.M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY, 1972.

[7] Peter M. Kogge, T. Sunaga, and e. a. E. Retter. Combined DRAM and Logic Chip for Massively Parallel Applications. In *IEEE Conference on Advanced Research in VLSI*, Raleigh, NC, 1995.

[8] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[9] Martin R. Linenweber. A Study in Java ByteCode Engineering with PCESjava. Master's thesis, Washington University in St. Louis, 2003.

[10] Sun Microsystems. javap - The Java Class File Disassembler. `http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javap.html`, 2001.

[11] David Patterson et al. Intelligent RAM (IRAM): Chips That Remember and Compute. In *IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[12] T.A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, CA, 1995.

[13] Stephen Shankland. Sun redials java for cell phones. `news.com.com/2100-1033-941504.html`, 2002.

[14] SPEC. Specjvm98 benchmarks. `www.spec.org/osg/jvm98`, 1998.

Vita

Christopher R. Hill

Date of Birth February 26, 1980

Place of Birth Orlando, Florida

Degrees Master of Science, Computer Science
Washington University in Saint Louis, May 2004

Bachelor of Science Magna Cum Laude, Computer Science
Washington University in Saint Louis, May 2002

Publications Lucas M. Fox, Christopher R. Hill, Ron K. Cytron, and Krishna Kavi. **Optimization of Storage-Referencing Gestures**. In *Proceedings of the ACM CASES '03 Workshop: Compilers and Tools for Constrained Embedded Systems*, October 2003

May, 2004