Report Number: WUCSE-2003-32

2003-05-16

Memory-Accessing Optimization Via Gestures

Lucas M. Fox

We identify common storage-referencing gestures in Java bytecode and machine-level code, so that a gesture comprising a sequence of storage dereferences can be condensed into a single instruction. Because these gestures access memory in a recognizable pattern, the pattern can be preloaded into and executed by a “smart” memory. This approach can improve program execution time by making memory accesses more efficient, by saving CPU cycles, bus cycles, and power. We introduce a language of valid gesture types and conduct a series of experiments to analyze the characteristics of gestures defined by this language within a set of benchmarks...
Read complete abstract on page 2.

Memory-Accessing Optimization Via Gestures

Lucas M. Fox

Complete Abstract:

We identify common storage-referencing gestures in Java bytecode and machine-level code, so that a gesture comprising a sequence of storage dereferences can be condensed into a single instruction. Because these gestures access memory in a recognizable pattern, the pattern can be preloaded into and executed by a “smart” memory. This approach can improve program execution time by making memory accesses more efficient, by saving CPU cycles, bus cycles, and power. We introduce a language of valid gesture types and conduct a series of experiments to analyze the characteristics of gestures defined by this language within a set of benchmarks written in Java and C. We gather statistics on the frequency, length, and number of types of gestures found within these benchmarks, using both static and dynamic analysis methods. We propose an optimization of the number of gestures required for a program, showing the optimization problem to be NP-Complete.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2003-32

Memory-Accessing Optimization Via Gestures - Master's Thesis, May 2003

Authors: Fox, Lucas M.

May 16, 2003

Abstract: We identify common storage-referencing gestures in Java bytecode and machine-level code, so that a gesture comprising a sequence of storage dereferences can be condensed into a single instruction. Because these gestures access memory in a recognizable pattern, the pattern can be preloaded into and executed by a ``smart'' memory. This approach can
improve program execution time by making memory accesses more efficient, by saving CPU cycles, bus cycles, and power. We introduce a language of valid gesture types and conduct a series of experiments to analyze the characteristics of gestures defined by this language within a set of benchmarks written in Java and C. We gather statistics on the frequency, length, and number of types of gestures found within these benchmarks, using both static and dynamic analysis methods. We propose an optimization of the number of gestures required for a program, showing the optimization problem to be NP-Complete.

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MEMORY-ACCESSING OPTIMIZATION VIA GESTURES

by

Lucas M. Fox

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 16, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

MEMORY-ACCESSING OPTIMIZATION VIA GESTURES

by Lucas M. Fox

ADVISOR: Dr. Ron K. Cytron

May 16, 2003
Saint Louis, Missouri

We identify common storage-referencing gestures in Java bytecode and machine-level code, so that a gesture comprising a sequence of storage dereferences can be condensed into a single instruction. Because these gestures access memory in a recognizable pattern, the pattern can be preloaded into and executed by a "smart" memory. This approach can improve program execution time by making memory accesses more efficient, by saving CPU cycles, bus cycles, and power. We introduce a language of valid gesture types and conduct a series of experiments to analyze the characteristics of gestures defined by this language within a set of benchmarks written in Java and C. We gather statistics on the frequency, length, and number of types of gestures found within these benchmarks, using both static and dynamic analysis methods. We propose an optimization of the number of gestures required for a program, showing the optimization problem to be NP-Complete.

to my family

Contents

List of Figures

Acknowledgments

I would like to thank my professor, advisor, and friend, Dr. Ron K. Cytron, for his introduction of the idea behind this thesis, in addition to his continuing feedback and encouragement as my work has progressed over the past three years; along with the National Science Foundation, which provided the funding for this research, and the other members of my committee, Dr. Mark Franklin and Dr. Jason Fritts.

I would also like to thank the various members of the Washington University DOC Group and general CS community who have contributed in one way or another to the environment I've been fortunate enough to work in, especially Ben Brodie, James Brodman, Dante Cannarozzi, Sharath Cholleti, Delvin Defoe, Steve Donahue, Scott Friedman, Matt Hampton, Mike Henrichs, Victor Lai, Nick Leidenfrost, Martin Linenweber, Stephen Torri, and the entire Monday Morning Basketball Squad; as well as Binny Mathews for his help with CommBench. I would especially like to thank Chris Hill, who has been through many of the trials and tribulations associated with this material himself, and has always been a valuable colleague in discussing anything from double indirections to double reverses.

Finally, I would like to thank Lisa, without whom I doubt any of this could have been possible.

Lucas M. Fox

Washington University in Saint Louis
May 16 2003

Chapter 1

Introduction

1.1 Processor in Memory

Gates and transistors have been rapidly decreasing in size for decades, freeing space on memory chips to be used for other tasks. One proposed method of utilizing this extra space is with a **Processor in Memory** (PIM), also known as **Intelligent RAM** (IRAM) [7, 10]. The Processor in Memory would be able to perform simple intelligent tasks independently and in parallel with the CPU. For example, a memory module would introspectively recognize its utilization and free portions of memory that were no longer needed by a program [2, 1]. The memory module could also perform simple processing tasks to decrease the load on the CPU. We propose using PIMs to execute series of related, memory-accessing instructions from a program within main memory[1]. Each intermediate instruction's result would *not* be returned to the CPU, only the final result of the series of instructions. Obviously, only certain instruction chains would be suitable for this task, so first we examine more closely what requirements are placed on this type of chain, which we will call a *gesture*.

1.2 Gestures

A **gesture**, also known as a **superoperator** [11], can be defined as a series of related machine instructions. This group of instructions is "related" in that only the end result of the series of the instructions is needed for further execution of the program; all the intermediate

[1] From this point on, the generic term "memory" will refer specifically to main memory. Please see §6.3 for information regarding the impact of cache.

instructions are simply steps that need to be taken towards the final result. In this sense, a gesture can be considered a single atomic step in the program execution. This type of chain may occur frequently within object-oriented programs, where programs often need to navigate through several layers of classes or structures in order to find a certain data item (*e.g.* `myname = ClassData.student.lastName`), or through a tree/list structure (*e.g.* `item = root.child.child`).

We are interested in series of instructions that read from and write to memory in some sort of recognizable pattern, in hopes that this pattern can be condensed into a single macro-instruction, which would be executed by the PIM. Specifically, the memory-accessing pattern of concern is one that dereferences[2] and "indirects" through memory addresses several times before returning a final result. For example, if we have container structures defined as

```
structure a {
  ...
  offset 4: x (pointer to a structure of type b)
  ...
}

structure b {
  ...
  offset 8: y (pointer to some other structure)
  ...
}
```

and our program accesses `a.x.y`, we would need to indirect through the memory address of `a` to find `x`, and likewise, through the memory address of `x` to find `y`.

That is, in each step of a chain of instructions, we can take the result of a the previous step's base-offset dereference and use it as the base address for a new base-offset dereference, where only the final dereference in the chain is needed for further execution. The entire gesture can be represented with three items, which are:

- The location of the memory address used as the base for the first dereference, which we term the **first source** of the gesture.

[2]The term "dereference" refers to the process of retrieving some value from memory using an indirect addressing scheme.

- The location to which the final dereference's result should be returned, which we term the **final destination** of the gesture.

- The series of offsets used in each dereferencing step, which we term the **indirection chain**.

For each step of a gesture, the result of the previous dereference is typically stored in a register or on a runtime stack. Because our goal is to consider an entire gesture as an atomic instruction, the intermediate steps of the gesture must be transparent to the rest of the program. Therefore, stored intermediate values must never be directly accessed by instructions that are not part of the gesture, and intermediate instructions must never change any program state that is not directly associated with the gesture.

Returning to our example of the gesture `a.x.y`, note that the pattern of dereferencing required to find the location of `y` is `(*(*(a+4)+8))`, where `*x` denotes the dereferencing of `x`. This is analogous to a series of low-level register-based memory fetching instructions shown below[3], of the form `GET(dest, source)`, where `source` is the address of the memory to be fetched, `dest` is where the result of the fetch should be written, and `RX` refers to the contents of a general purpose register numbered `X`:

```
--unrelated instructions--
GET(R2, R3+4)
GET(R4, R2+8)
--unrelated instructions--
```

Note that only the result in `R4` is important to the execution of the program, and that it only depends on the value that was in `R3`, along with the successive indirection offsets 4 and 8. Thus, we can think of the first two lines of the above example as a single gesture, which takes the value in `R3`, indirects from memory twice with the offsets 4 and 8, and returns the final result to `R4`. A diagram of the interaction between the CPU and memory that occurs during execution of this gesture is shown in Figure 1.1.

[3]From this point on we will represent the series of instructions that constitute a gesture as a vector $< Instruction1, Instruction2, ... >$ in order to save space. For example, the gesture shown here would be represented as `<GET(R2, R3+4), GET(R4, R2+8)>`.

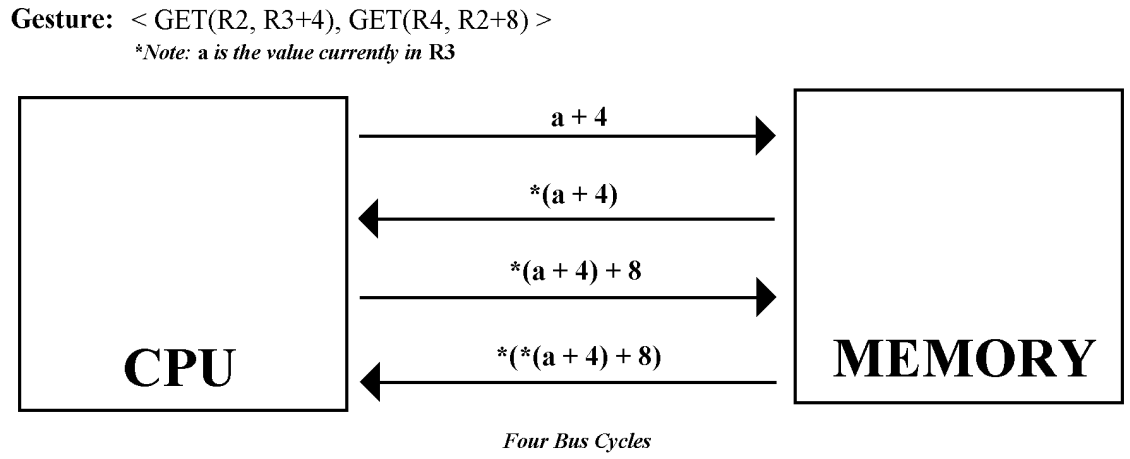

Figure 1.1: Interaction between CPU and Memory during a Gesture

1.3 Memory Macros

If we can determine what memory-accessing gestures will be used in the course of a program's execution, we can encompass the execution of these gestures within a **memory macro**, which will store the relative offsets of each memory accessing instruction in the gesture. These macros can then be interpreted and executed within the PIM in lieu of executing the entire chain of instructions on the CPU. Specifically, we can assign a unique macro number to each unique indirection chain that is found in the program, and then encompass that number within a new instruction type that is recognized by intelligent memory. The mapping of each chain to its corresponding macro number will be loaded into memory before the program executes and stored in a **macro table** that the PIM can access. Note that we only preload the indirection chain for a gesture on the PIM, so that the same macro can be used for different first source / final destination pairs that share the same series of offsets.[4]

During execution, when the CPU encounters a gesture, instead of sending a series of standard memory-accessing messages, it will make one macro call to the PIM, containing the first source, final destination, and indirection chain (macro) number. For example, instead of sending the series of fetch instructions described at the end of §1.2, we would simply send the macro instruction `GET_MACRO_4(R4,R3)`, where `GET_MACRO_4` had

[4]Due to the one-to-one relationship between an indirection chain and a macro, we will use indirection chains, in the format of the tuple (first indirection offset, second indirection offset, ...), to identify particular macros in the following sections.

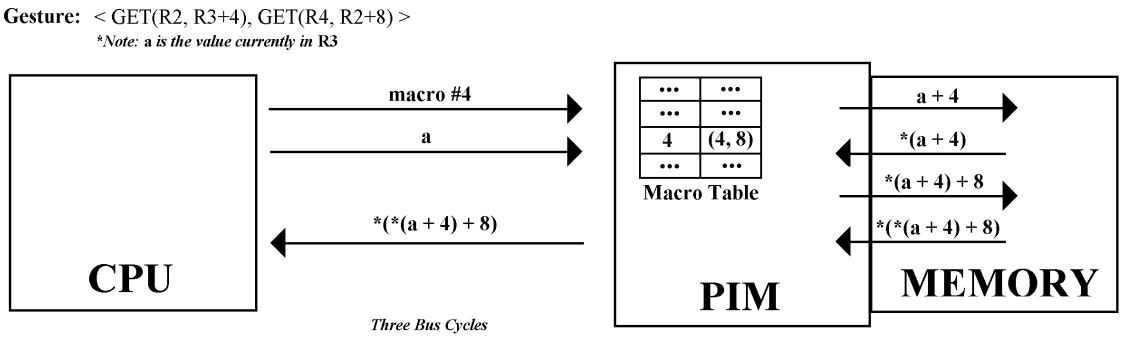

Figure 1.2: Interaction between CPU and Memory during a Gesture, with Memory Macros

previously been defined to the PIM as a double indirection macro with offsets of 4 and 8 for the first and second indirections, respectively. Upon receiving one of these instructions, the memory can then find the macro corresponding to the macro number defined in the instruction, and then use that macro to execute all the steps of the indirection chain within the memory, returning only the final result to the CPU. Figure 1.2 demonstrates the interaction between the CPU and memory that occurs during the execution of a gesture with a memory macro.

1.4 Potential Benefits

The benefits of our approach are as follows:

CPU Cycle Savings: If all the instructions in a gesture can be executed by the PIM, the CPU does not have to execute them, and is free to do other processing while waiting for the gesture's final result to be returned. Thus, the load on the CPU can be reduced, increasing throughput. This idea can be most effectively exploited when zero-latency multithreaded processors are employed.

Bus Cycle Savings: If all the instructions in a gesture can be executed by the PIM, we do not need to send an address to memory and send a resultant value back to the CPU for each step of the gesture. Thus, we eliminate a certain percentage of bus cycles in those systems where the bus is shared.

Power Savings: Each bus cycle requires power to recharge the bus. Therefore, if we reduce the number of bus cycles, and the subsequent reduction in power is less than

the extra power needed for our memory logic and macro table, we can reduce overall power requirements.

Decreased Footprint Size: Since our system would use a single macro instruction to represent the entire chain of instructions composing a gesture, we reduce the total number of instructions in a given program.

1.5 Goals

It should be clear that all these benefits are directly related to the number of gestures that occur within a program. Programs with a higher gesture frequency should see increased savings in cycles, time, and power. Therefore, our primary goal in the series of experiments presented in Chapter 5 is to determine how often gestures occur in programs, so that we can determine the potential savings. We will also investigate the distribution of gesture lengths within a program, as longer gestures would likely be more beneficial, as well as the number of distinct gestures in a program, which would be related to the number of needed macros.

1.6 Techniques

We will make use of two techniques to gather our information:

Static Analysis: Examination of a program independently of its execution, which would be necessary if we are going to send memory macros to the PIM before execution starts.

Dynamic Analysis: Examination of a program during execution, which would provide the most accurate picture of how often gestures occur in a program. This provides an upper bound on the number of gestures that could potentially be found with static analysis.

We concentrate our gesture-finding experiments primarily on programs written in Java^{TM}, for several reasons:

- The intermediate representation of a Java `.class` file provides an excellent way of accessing the bytecode of a program in a static context.
- The software-based implementation of the **Java Virtual Machine** (JVM) provides a good platform for instrumentation pertaining to dynamic bytecode analysis.

Further benefits of using Java will be discussed later. However, to provide a broader perspective, we also conducted a simple examination of potential gestures in C programs, by instrumenting a system software infrastructure.

Chapter 2

Gesture Types in Java

Before we present the experiments conducted to find gestures in Java programs, it is useful to define what type of gestures we expect to encounter. Within our experiments, we investigate gestures that occur on three types of Java bytecode instructions: `getfield`, `putfield`, and `getstatic` [8]. We describe these instructions and the gesture classes associated with each of them individually, and then define a simple language of valid gesture types using these instructions.

2.1 Getfield Gestures

The simplest gesture type is a series of consecutively-executed `getfield` instructions. A `getfield` is used in the JVM specification to fetch a field at a specific field index of a specific class. In this instruction type, the top of the stack, which is required to contain an object reference, is popped, and the value of the specified field in that object is pushed onto the stack, as shown in Figure 2.1. If we chain these instructions together, the next `getfield` instruction will then take the value that has just been pushed and use it as a new object reference, and so on. We will term this series of getfield instructions a **getfield chain**.

Because all Java bytecode instructions, including `getfield`, perform a well-defined operation with respect to the stack, we always will know the state of the stack as the next instruction in a gesture is executed. For each `getfield` instruction, the stack size does not change, and only the top item on the stack is modified. The intermediate steps of the `getfield` chain do not influence the final stack state because they are successively overwritten until the final result of the gesture is put on the stack. Hence, the top of the stack

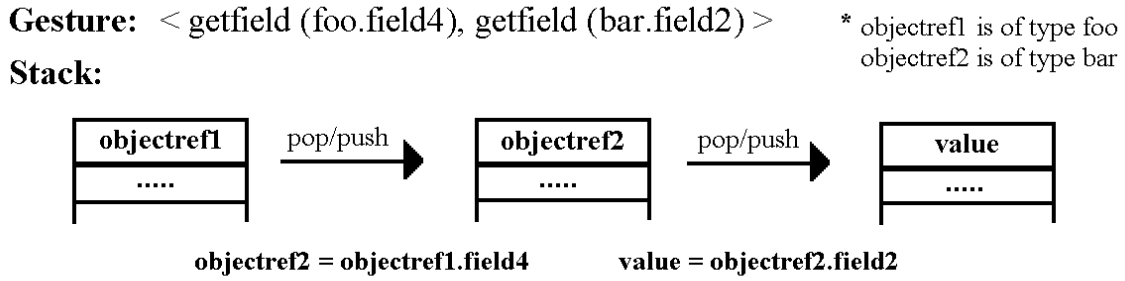

Figure 2.1: Getfield Gesture Stack Operations

acts as the first source and final destination for the gesture, by initially holding the source address, and then the returned result.

Our macro for a particular gesture then consists of the indices of the fields to be retrieved in the object references that are encountered on each step of dereferencing the `getfield` chain. To "call" this macro, the program would send the original object reference from the top of the stack (which acts as the first source and final destination), along with a macro number, as was described in §1.3.

We can obtain the series of indirection offsets independently of the program's execution because they are specified in the bytecode instructions, as indices into the constant pool. The relative ease with which we can extract the first source, final destination, and indirection chain for this type of gesture makes it an excellent candidate for incorporation within a memory macro.

2.2 Getfield-Putfield Gestures

The second gesture type we investigated is very similar to the `getfield` chain mentioned above. This gesture consists of one or more consecutive `getfield` instructions followed by a single `putfield` instruction. We will term this type of gesture a **getfield-putfield chain**. As would be expected, `putfield` does the complement of a `getfield` and sets a specified field of an object. More precisely, the `putfield` interprets the top value on the stack as the value to be stored and the second value on the stack as a reference to the object in which the value will be stored (see Figure 2.2).

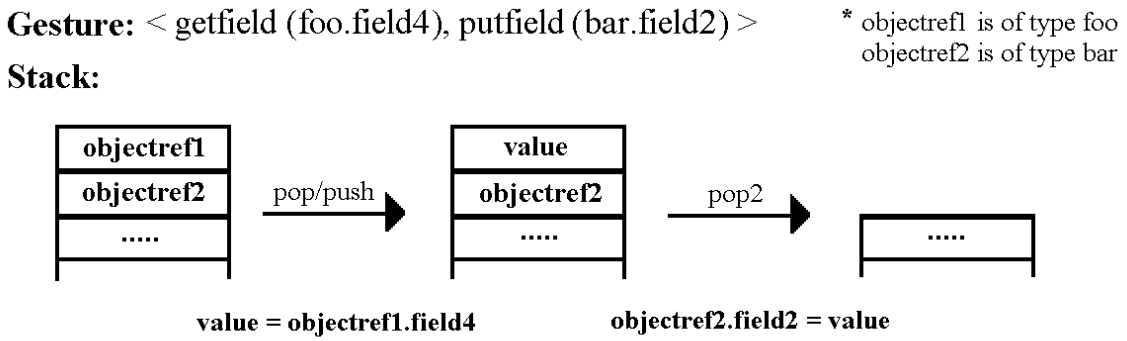

Figure 2.2: Putfield Gesture Stack Operations

If the `putfield` is proceeded by a series of `getfield`s, then the value to be stored (the one at the top of the stack) will be the result of that `getfield` chain, so the entire gesture can be considered atomic. The key difference between this gesture and a `getfield` chain is that this gesture ends with a `store` instruction, which means the final result of the instruction chain never has to be returned to the CPU. Thus, this gesture has more potential benefit than a standard `getfield` chain, because it frees up an additional bus cycle and does not force the CPU to wait for a result.

A `putfield` macro would then consist of a chain of `getfield` offsets and a final offset of the field within the object where the result of the gesture would be stored. Like the `getfield` offsets, this field offset can also be found in the constant pool. To "call" this macro, we again need the original object reference and the macro number, as was the case for the `getfield` macro. We also must include a reference to the object where the final value should be stored, which is no longer simply the top of the stack, but a specified location in memory. This final object reference can be resolved as the second item from the top of the stack.

2.3 Getstatic-Getfield Gestures

In the process of developing our more advanced gesture-finding programs, it was determined to add an additional bytecode type to our definition of what would constitute a valid gesture. This bytecode, the `getstatic` instruction, can only validly occur at the beginning of a gesture, much like the `putfield` instruction can only occur at the end.

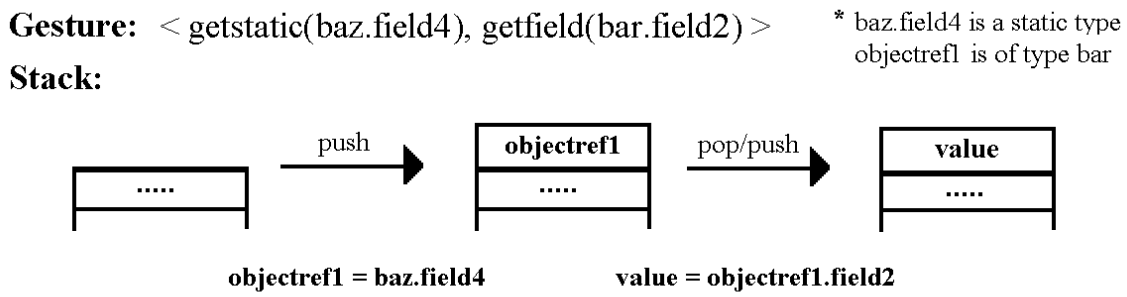

Figure 2.3: Getstatic Gesture Stack Operations

`getstatic` pushes a static field from a class onto the stack, but because the field is static, no object reference is required for this instruction (see Figure 2.3).

It would not make sense to chain `getstatic` instructions together since they fetch based only on the class and field index, not the current state of the stack, so there is no inter-relation between each `getstatic` instruction. However, the fetched `getstatic` value could subsequently be used as an object reference for one or more `getfield` instructions that follow, which do not increase or decrease the stack size. Thus, we can define two new types of gestures by attaching a `getstatic` instruction to the beginning of each of the previously defined gestures. We characterize these new gestures as **getstatic-getfield chains** and **getstatic-putfield chains**. Macros for these types of chains are very similar to their previously-defined equivalents, only they use a constant pool index when invoking the macro rather than an object reference from the top of the stack.

2.4 A Language of Java Gestures

Based on the discussion above, we have four gesture types, which can be defined by the following regular expressions over the alphabet $\Sigma = \{g, p, s\}$, where g represents a `getfield` instruction, p represents a `putfield` instruction, s represents a `getstatic` instruction:[1]

[1]For our formal language definitions we will use the notation described by John Martin in *Introduction to Languages and the Theory of Computation* [9].

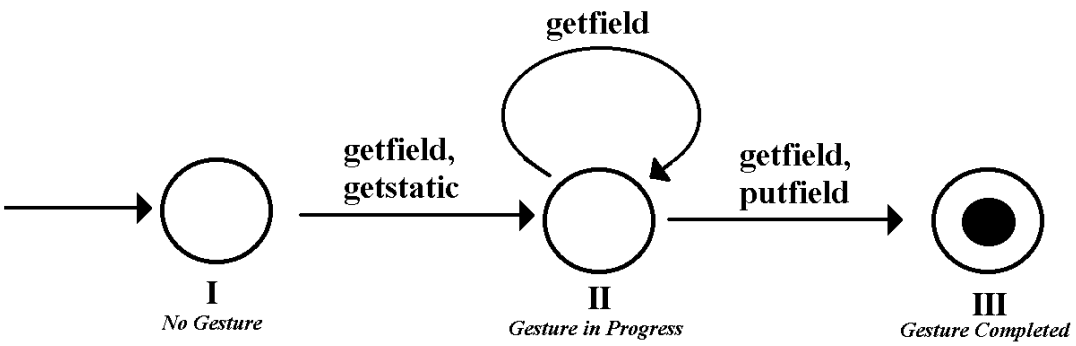

Figure 2.4: A Gesture-Recognizing NFA

1. $[gg^*g]$ *(Getfield Gestures)*
2. $[sg^*g]$ *(Getstatic-Getfield Gestures)*
3. $[gg^*p]$ *(Getfield-Putfield Gestures)*
4. $[sg^*p]$ *(Getstatic-Putfield Gestures)*

or, simply $GEST = [(s + g)g^*(g + p)]$.[2]

If we think of the series of instructions in a Java program to be analyzed as a string i over the alphabet $JB = \{x \mid x \text{ is a Java bytecode}\}$, the problem of finding gestures in a program is equivalent to finding the subsets of $GEST$ that occur as a substring of i:

$$FOUND = \{g \in GEST \mid \exists x, y \in JB^* \; xgy = i\}$$

Similarly, if we think of $FOUND$ as a multiset, it would contain every occurrence of any gesture that is found in the Java program's series of instructions. From this set we can obtain not only a list of which gestures have been used, but also the frequency of each gesture.

2.4.1 A Gesture-Recognizing NFA

Like all regular languages, $GEST$ can be recognized by a **Nondeterministic Finite Automaton** (NFA), as is shown in Figure 2.4. Although none of the gesture-finding programs used in the experiments are specifically modeled after this NFA's explicit "states", their basic structure often mirrors the machine's fundamental components. In particular, we do include concepts of a gesture-starting and gesture-ending state, and symbols that we are allowed to include next in a gesture are based on keeping track of what we have already seen.

[2] In some experiments we will attempt to find only a subset of this language (*e.g* the Getfield Gestures) for simplicity's sake.

Chapter 3

Reducing the Number of Memory Macros

Now that we have defined the characteristics of a set of Java gestures, let us examine how these gestures would then be encompassed in a memory macro.

3.1 Simple (Non-Reordered) Macro Generation

The simplest way to generate memory macros for a Java program would be to scan the program's `.class` files using a static analysis method (see §5.2) to find all the gestures it uses, then determine which macros will be needed to represent all the gestures we have found. For example, if our set of classes and their corresponding fields were defined as they are in Figure 3.1, and the program contained the following list of instructions:

```
a = new Foo();
b = new Bar();
c = new Baz();
...
Bar temp1 = a.x.y;
Foo temp2 = a.y.x.x;
Foo temp3 = b.x.y;
Bar temp4 = c.y.y;
```

then the four gestures found through our analysis program would be:

	Class `Foo`	**Class** `Bar`	**Class** `Baz`
Field 1:	`Bar x`	`Foo x`	`Foo x`
Field 2:	`Foo y`	`Bar y`	`Bar y`
Field 3:	`Bar z`		

Figure 3.1: Example Field Alignment

```
1) <getfield (Foo.field1), getfield (Bar.field2)>
2) <getfield (Foo.field2), getfield (Foo.field1), getfield (Bar.field1
3) <getfield (Bar.field1), getfield (Foo.field2)>
4) <getfield (Baz.field2), getfield (Bar.field2)>
```

We would need three memory macros, one for each of the indirection chains (1,2), (2,1,1), and (2,2). Each of these macros would then be assigned a number, and the mapping of macro numbers to indirection chains (*e.g.* `MACRO_1`→(1,2), `MACRO_2`→(2,1,1), `MACRO_3`→(2,2)) would be sent to the PIM before program execution. Then, during execution, the CPU would tell the PIM to execute one of these macros at the point when the first `getfield` of the corresponding gesture would have been called.

Notice here how we do not necessarily need a separate memory macro for each gesture we encounter, even if the gestures operate on different classes. If the series of offsets (*i.e* the indirection chain) for a group of gestures is identical, we can state that they can be **covered** by the same macro. Unlike structures in C or C++, all non-primitive fields in a Java class are the same size because they are all considered references[1]. This means that any field at a specific index can be accessed the same way as any other field at that index, regardless of which object or class each field is in. Since a field lookup is based only on the offset of the field index within a Java object, the class of the object is irrelevant from the perspective of the `getfield` instruction, only the index itself is significant. Therefore, if this type of instruction is covered by a macro, we can ignore the classes of objects in a gesture, as long as the sequence of field indices is the same. Hence, gesture one and gesture three in the above example are covered by a single macro, (1,2).

[1]From this point on we will assume that all fields are reference fields, unless otherwise noted, in order to simplify our discussion.

3.2 Field Reordering

As was shown above, the number of macros needed to cover all gestures can be fewer than the number of unique gestures encountered. Macro information must be stored in main memory, so any reduction in this number reduces the amount of memory overhead that a macro-processing system would require. Thus, it makes sense to attempt to reduce the number of needed macros. Specifically, if we have a macro corresponding to some indirection chain (x,y) it would be beneficial to put as many gesture-referenced fields as possible in the x^{th} and y^{th} positions of their class files, since only the field indices on each step of a gesture's indirection chain determine whether or not it will be covered by a macro.

Fortunately, the inherent referential transparency of Java allows us to do just this, by assigning any ordering we want to the fields of a class, if they are all the same size. The order of fields that the programmer specifies is independent of the order they are presented in the `.class` file, because Java fields cannot be accessed directly though offsets, as is the case in C/C++ structures; they can only be accessed through their proper names.

Consider reduction of the number of macros needed in the above example. If we change the order of the fields in class Baz as follows:

```
Class Baz {
  field 2: Foo x;
  field 1: Bar y;
}
```

then gesture number four becomes

```
< getfield (Baz.field1), getfield (Bar.field2) >
```

This gesture can then be covered by the macro (1,2), which already covers two other gestures, so the total number of macros has been reduced from three to two.

However, complications arise in the process of reordering due to the fact that different gestures may reference different fields of a class. Because of this, rearranging (*i.e.* permuting) the order of fields in a particular class so one gesture conforms to a macro may cause other gestures referencing the class to need a different macro. Permuting one class in an effort to decrease the macro count could potentially increase the total number of macros needed. Thus, the problem of finding a reordering scheme that minimizes the number of macros could be very difficult. In Chapter 4 we show that finding the minimum number of macros needed to cover some group of a program's gestures is NP-Hard, and would likely take exponential time.

Class.Field	1st indirection	2nd indirection	3rd indirection
`Foo.x`	1	1	0
`Foo.y`	1	1	0
`Foo.z`	0	0	0
`Bar.x`	1	0	1
`Bar.y`	0	2	0
`Baz.x`	0	0	0
`Baz.y`	1	0	0

Figure 3.2: Example Gesture Reference Frequency Table

3.3 A Greedy Reordering Heuristic

Given the apparent difficulty of trying to minimize the number of macros, heuristics could be used to reduce the number of macros. The greedy heuristic reordering algorithm used in our experiments was introduced by Chris Hill[4] and makes use of a few intuitive rules for field reordering:

- We can never make things worse by moving all fields that aren't part of any gesture to the end (*i.e.* the highest field indices) of each class. Their order is irrelevant since we don't need to "cover" them, and by forcing all referenced gestures into the lowest field indices we increase the chance of one macro covering several instructions.
- We can move any primitive fields near the end of the class since they can only be referenced by the last step of a gesture. Of the fields that are referenced by gestures, we can move those that are referenced most frequently to the lowest field indices. This allows the greatest number of gestures to be covered by a few low-number macros (*e.g.* (0,0), (0,1), (1,0), etc.), and leaves only the infrequently occurring gestures to be covered by higher macro numbers.

To take advantage of these ideas, we can construct the following reordering algorithm:

- Construct a gesture reference frequency table from the list of gestures found in a program, which contains the number of times each field in each class is referenced by each indirection step in any gesture. Thus, the table will have a maximum of

$$(maximum\ fields/class) * (number\ of\ classes) * (maximum\ gesture\ length)$$

items.

- Move all fields that are not referenced in any chain to the highest field indices of the class, in any order.
- Of the fields that are found in a chain, move all reference (*i.e.* non-primitive) fields to lower field indices than any primitive field index.
- For each class in the table, order reference fields within their block of indices based on the number of times they are referenced by the first indirection step of a gesture. Those that are referenced most frequently get the lowest field indices, and vice versa. To break ties, use the number of times the field is referenced by the second indirection step of a gesture. Continue breaking ties using the next indirection step of the gesture until there are no more ties or until all indirection steps have been used.
- Order primitive fields within their block of indices in each class in this same manner, once all reference fields have been ordered.

In our example above, the gesture reference frequency table would be constructed as is shown in Figure 3.2. Then, following our algorithm, the only fields that are not referenced by any gesture at any indirection step are field `z` in foo and field `x` in `Baz`. These would be moved to the highest field index of each class; `Foo.z` remains at field index 3, while `Baz.x` moves to field index 2, forcing `Baz.y` to move to field index 1. Notice that we have already reduced the number of needed macros, by performing the same permutation that was described in §3.2. Once this permutation has been carried out, all the fields in all the classes have been ordered successfully according to the algorithm and we are done. Notice that `Bar.y`, even though it has a greater count than `Bar.x` for the second indirection step and the same total indirection count, is not switched because we always begin by examining the first indirection step, where `Bar.x` does have a higher count than `Bar.y`.

Use of the heuristic described above is able to reduce the number of needed macros in many cases, but does not guarantee an optimal ordering to fields.[2] The following counterexample describes a situation in which the heuristic does not give an optimal solution.

Suppose we are given a group of classes with the field alignment specified as in Figure 3.3. Then, if our program contains the following set of instructions:

[2]Additional testing performed by Chris Hill [4] has shown that our heuristic does in fact generate the minimum number of macros needed, in many cases.

	Class `Foo`	**Class** `Baz`	**Class** `Top`	**Class** `Bot`	**Class** `Bar`
Field 1:	`Top a`	`Bot d`	`int f`	`int k`	`Bot m`
Field 2:	`Top c`	`Bot e`	`int g`	`int l`	

Figure 3.3: Counterexample Field Alignment

Class.Field	**1st indirection**	**2nd indirection**
`Foo.a`	2	0
`Foo.c`	1	0
`Baz.d`	2	0
`Baz.e`	1	0
`Top.f`	0	2
`Top.g`	0	1
`Bot.k`	0	2
`Bot.l`	0	1
`Bar.m`	1	0

Figure 3.4: Counterexample Gesture Reference Frequency Table

```
int temp1 = Foo.a.f
int temp2 = Foo.a.g
int temp3 = Foo.c.f
int temp4 = Baz.d.f
int temp5 = Baz.d.k
int temp6 = Baz.e.l
int temp7 = Bar.m.k
```

the set of macros needed to cover all these instructions given the current field ordering would be $\{(1,1),(1,2),(2,1),(2,2)\}$. Applying our heuristic, we would next construct the gesture reference frequency table shown in Figure 3.4. Based on this table, no field reordering is done, so we remain with four macros after using our heuristic.

However, if we permute fields of class `Bot` such that `k` is now the second field and l is now the first, our set of needed macros would become $\{(1,1),(1,2),(2,1)\}$. Thus, the number of needed macros after application of our heuristic is not optimal, because we were able to find a better solution through inspection.

Chapter 4

Complexity of an Optimal Field Reordering Algorithm

As has been stated in §3.2, achieving a reordering of fields that will result in a minimum number of memory macros appears to be a very hard problem. In this chapter we will attempt to more precisely describe the difficulty of this problem, by showing it belongs to a class of problems that are NP-Hard. Specifically, we will show that finding the minimum number of macros needed to cover some group of a program's gestures is NP-Hard. First, however, let us formally define the problem of field reordering in mathematical notation, which will be more conducive to our proof.

4.1 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 n^{th} 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 (*i.e.* m is analogous to the indirection chain as we have defined it in §1.2).

- 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 j^{th} 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.

4.2 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

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

4.3 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 that problems can be categorized in. **P** is used to represent the class of problems that can be solved in polynomial time, whereas **NP** is used to represent 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, we must 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, that 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 out of an arbitrary instance of a known NP-Complete problem and showing that a solution to the NP-Complete problem must also be a solution to our problem as well as that a solution to our problem must also be a solution to the NP-Complete problem [5].

4.4 Theorem 1 Proof

4.4.1 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 one for any type in T.[1]

4.4.2 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.

4.4.3 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 $s(a) \in N^+$ for each $a \in A$, and a positive integer B.

Let us 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 a 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:

[1] Obviously, the problem is trivial if all types have only one field.

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 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[2] 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]From this point on, $\bigcup$ will always denote a multiset union unless otherwise noted.

4.4.4 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

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

4.4.5 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 can be expressed as:

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

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:

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

which means a solution to the subset problem exists.

Since we have now proved 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. ■

4.4.6 Example

Because we have a unique set of cloned instructions for each type, no two groups of cloned instructions are 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 indirection chain we wanted to cover was (1,0), its corresponding type vector τ was <`foo bar baz`>, and our macro was (0,0); 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 $a \in A$. 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

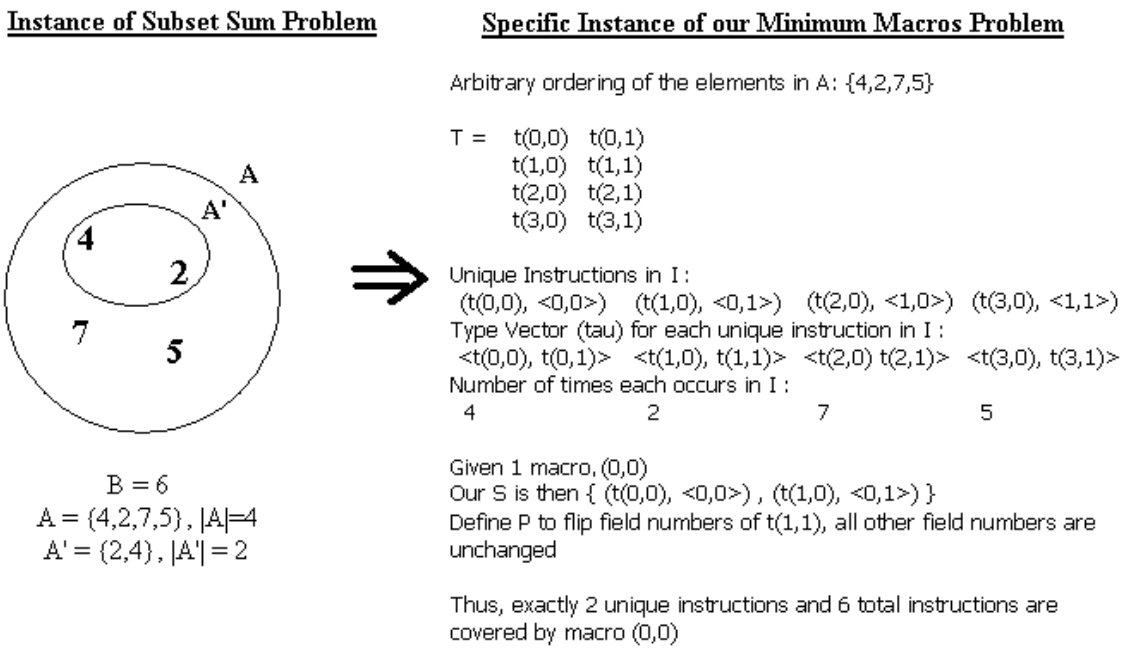

Figure 4.1: Pictorial Example of NP Reduction

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.

Another example of an NP reduction is shown in Figure 4.1.

4.5 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 $|M| = 1$. ■

4.6 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. ■

4.7 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 §4.3, 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. ■

4.8 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 $D5(P, \mu)$ is NP-Complete, where $D5(P, \mu)$ is defined as follows:

$$D5(P, \mu) \ \ (\exists\ S \subseteq I)\ (\exists\ \rho)\ |C(\lambda(S,\ \rho),\ M)| = |I|,\ |M| \leq \mu$$

The optimization form of this problem is then to find:

$$Opt(P) = \min_{n \in N} D5(P, n)$$

The difficulty of trying to use our reduction from Subset Sum to prove this conjecture arises from the fact that the Subset Sum problem becomes trivial when we force it to use the entire provided set of integers. However, we believe that our problem continues to

be NP-Hard even if we were to include the entire set of instructions. In development of the heuristic reordering strategy defined in §3.3, we have operated under the assumption that this conjecture is true and there is no polynomial optimization algorithm.

Chapter 5

Experiments

5.1 Experiment Aims

A series of experiments was carried out to determine how frequently memory-accessing gestures occur in real programs, and to investigate the properties of those candidates. Specifically, we are interested in questions about gesture properties such as:

- **How often do gestures occur in the context of a `.class` file?**
- **How often do gestures occur in running programs?**
- **What is the distribution of gesture length in a program?**
- **How many different gestures are there in a program?**
- **How can the number of gestures used in a program be optimized?**

A group of benchmark programs, written in one of two higher-level languages, Java and C, were analyzed to answer these questions. The suitability of different candidate gestures, and therefore the analysis technique we use in finding gestures, depends on the type of machine on which the program code is being executed. A stack-based machine, such as the JVM that the Java programs run on, has different requirements for defining gestures than a register-based machine, such as the SimpleScalar Architecture that the C programs run on, because of differences in the way each system stores program state. Three tools were used in the analysis of Java programs and one tool was used to analyze programs in C. In the following sections, we discuss each of these tools in detail. We also discuss the Memory Macro Simulation package, which helps us reexamine our results from the other

tools in order to better understand the performance gains that could be achieved through use of memory macros.

5.2 Static Analysis Methods

Two tools, javap and Scavenge, were used to find gestures in Java `.class` files using a static approach. All static gesture-finding programs share certain characteristics, which we now discuss before presenting the specific features of each tool.

5.2.1 Benefits and Limitations of Static Analysis

The chief advantage to static analysis tools is, quite simply, that they do not require execution of the program to gather their data. All static analysis tools present a sequential and direct mapping of the `.class` file's contents, which means that searching for gestures is basically a linear search through these files. Therefore, static analysis times for `.class` files are bounded primarily by the size of the file, not the execution time of the program that the file is a part of. Thus, we can spend significantly less time analyzing a long-running program under static analysis than we would under dynamic analysis. Static analysis is a particularly useful tool for determining memory macros, where we need to know all the gestures that will be encountered in a program before it executes, so they can be loaded into the PIM beforehand.

However, limitations also arise from the fact that the `.class` files do not trace any execution path through the program, simply presenting an in-order display of the bytecode layout for each method described in the file. Because of this, these tools cannot find the true frequency at which gestures occur. For example, there is no easy way of determining whether a particular bytecode is executed once or in the context of a loop that may execute 1000 times. In addition, these static approaches are bound to individual methods and therefore cannot recognize gestures that span across method boundaries.

5.2.2 Static Gesture Candidate Selection

Static analysis methods allow us to gather several useful types of information. First, we can determine whether or not a given gesture exists within a class or, similarly, determine the list of all unique gestures that occur within a class. Here, we do not care how many times the gesture may be executed, so we can simply examine the code section of a `.class`

file without worrying about how the program actually executes. This examination can be accomplished with a sequential scan of the class file, where we record any sequences of instructions that match our set of valid gestures, thereby generating a list of all the gestures a program uses. The process is basically one of regular language recognition, where our set of valid gestures can be thought of as the set of substrings in our sequence (*i.e.* string) of valid instructions that match some regular expression (see §2.4). Therefore, our code is quite simple, and primarily concerned with keeping track of the **gesture state** at each step of processing the instruction sequence. This gesture state would tell us whether or not the current instruction could be part of a gesture, and if so, the gesture's type and current size.

This approach also allows us to determine the maximum-length gesture that occurs in a given class. Like the previous procedure, this procedure also does not depend on the number of times a gesture is executed. To determine the maximum-length gesture, we simply keep track of our current gesture length while sequentially scanning the `.class` file.

Finally, static analysis can also be used to count the number of places gestures occur in the `.class` file, but this does not directly correspond to the number of places gestures are executed. Nevertheless, in practice, these results did roughly mirror results from using more sophisticated, dynamic techniques. Therefore, this approach could be used to at least give a rough estimate of the number of chains executed, because it involves far less computation than dynamic analysis.

5.2.3 Benchmarks

To test static gesture recognition ability, and to determine if gestures do in fact occur in some industry-grade Java programs, we applied our tools to the **SPECjvm98 Benchmarks** [13]. Specifically, SPECjvm98 contains eight different test applications, five of which are either real applications or derived from real applications that are commercially available. In these experiments, we are chiefly concerned with the `.class` files that are provided for each application, which we can scan for gesture candidates. A brief description of the eight applications follows:

_200_CHECK: A simple program to test various features of the JVM to ensure that it provides a suitable environment for Java programs.

_201_COMPRESS: Implements a modified Lempel-Ziv compression method (LZW). It basically finds common substrings and replaces them with a variable size code.

_202_JESS: The Java Expert Shell System, which continuously applies a set of rules to a set of data to solve a set of puzzles.

_205_RAYTRACE: A raytracer that renders a scene depicting a dinosaur.

_209_DB: Performs multiple database functions on a memory resident database.

_213_JAVAC: The Java compiler from the **Java Development Kit** (JDK) 1.0.2.

_222_MPEGAUDIO: An application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification.

_227_MTRT: A multi-threaded version of _205_RAYTRACE.

_228_JACK: A Java parser generator.

5.3 Static Analysis With Javap

The first tool we used in our analysis, **javap** [15], is a simple `.class` file disassembler tool. This utility, when the -c option is invoked, can be used to take a `.class` file as input and display the disassembled code. In particular, it displays the Java instructions that correspond to each bytecode for each method in the class, providing a simple approach to answering questions one and four (and three, to a lesser extent) of §5.1, through the methods described above.

5.3.1 Implementation of Javap

Two wrapper scripts were used to apply the javap tool to the SPECjvm98 benchmark. First, a C shell script was used to find all the `.class` files for each benchmark. Second, a Perl script was used to take this list of `.class` files, run `javap -c` on each, and determine the number and length of `getfield` chains that occur in each group of files. Because the limited information provided by javap forces this approach to emphasize simplicity, classes were scanned on a file-by-file, not class-by-class basis, and only `.class` files in the program's directory were scanned. For example, any classes from the Java API that were used by a benchmark was not scanned. As an attempt to fill in some of the missing information about the Java API, all of the classes from each of the first-level Java API Packages were scanned independently in a manner similar to the benchmarks.

5.4 Static Analysis with Scavenge

Scavenge was developed as a static analysis tool to address some of the limitations that were encountered in §5.3, and to specifically investigate the number of macros that would need to be loaded onto a PIM for a given program.

The chief advantage that Scavenge has over the javap approach in gesture-finding capability is that it can scan *all* the `.class` files used by the program, including those from the Java API, by recursively examining the current classpath and the classes referenced within each `.class` file. However, because Scavenge is still limited to static analysis, it is prone to the same failings as were described in §5.2.1. In addition, Scavenge processes at a slower rate than javap, since it is a true Java application rather than a simple script which scans an output file.

Like javap, Scavenge allows us to find information relating to questions one, three, and four of §5.1, by providing information about the gesture length and the frequency at which different-length gestures occur in each `.class` file. However, unlike javap, Scavenge also provides a way to answer question five, because it implements a gesture-reordering algorithm (see §3.2). Therefore, we can use also use Scavenge to make a more sophisticated estimate of the number of types of gestures that will be executed (corresponding to the number of macros that will be needed) for a given program, beyond simply counting the number of unique gestures.

5.4.1 Implementation of Scavenge

The set of benchmarks described in section §5.2.3 were again used for this set of experiments. However, because Scavenge has the capability to find and scan all classes used by a program, including those in the Java API, we can conduct a more thorough analysis and obtain more accurate results than were found in §5.9.1.

Scavenge is written in Java and utilizes the **Jclasslib** [3] library, which provides an interface with the underlying components of a `.class` file, allowing developers to read, write, or modify it. Here, this library is used to gain access to the disassembled code portion of each `.class` file that is used by the program for gesture-counting purposes, in a manner similar to javap. We can also use Jclasslib as a means to determine which `.class` files are associated with a given program, because the library contains functions which can tell us the type of each object that is created in the program's code. With this information, if we find a new object of a type that we have not previously encountered, we know to recursively scan the `.class` file associated with that object for any additional gestures.

In its simplest form, Scavenge iteratively scans each `.class` file for consecutive `getfield`s in a manner similar to that described in §5.2.2 to determine statistics such as the maximum gesture length and number of unique gestures. Based on the indirection chain of each recognized gesture, Scavenge also determines how many and what type of memory macros would need to be loaded into the memory unit (see §1.3).

Because Jclasslib allows us to draw an accurate picture of which gestures occur in a program, we were able to add the additional functionality of a field reordering algorithm to Scavenge. Scavenge has the option to change the field layout of any `.class` file it encounters in an effort to reduce the number of unique gestures and thus the number of macros needed for a program, as was described in §3.2. We know that the problem of reordering fields in such a way that would result in a minimal number of macros is NP-Complete (see Chapter 4), so we use the reordering heuristic that is described in §3.3. After reordering the fields of all classes used by the program according to this heuristic, Scavenge outputs the number of unique memory macros that would need to be loaded onto the PIM with and without this reordering.

5.5 Dynamic Analysis

As was demonstrated in §5.3 and §5.4, static analysis of gestures has its limitations. Most importantly, it is very difficult to determine how many times a gesture occurs in the course of a program's execution. Essentially, we have not yet addressed how we would go about *accurately* answering questions two and three of §5.1. The natural solution to this problem is to examine bytecodes as they are executed in the context of a running program, with dynamic analysis methods.

5.5.1 Dynamic Benefits and Limitations

Although we can do a good job of statically determining how many unique gestures there are in a program, a dynamic approach is needed to determine each of these gestures' relative importance, by finding the distribution of their lengths and the frequency at which they occur. In addition, with dynamic analysis we can find gestures that span method boundaries, eliminating another problem of static approaches.

One may wonder why dynamic analysis should be used if the memory macros corresponding to recognized gestures would need to be loaded into memory before the program is executed. The primary goal of this dynamic approach is to determine an *upper bound*

on the number of gestures that could be executed by a program, to determine whether or not these gestures occur frequently enough to merit further investigation, and to determine what type of gestures occur most frequently. The dynamic approach described here is not intended to be integrated into whatever final software package processes the memory macros that are needed for a program; it is specifically designed to gather information. Therefore, this approach emphasizes a thorough analysis of the program at the expense of increased computational time and complexity.

5.6 The Dynamic Gesture Searcher

In order to accomplish the task of dynamically analyzing Java programs, we have instrumented the JDK 1.1.8 [14] to form an analysis tool called **DYnamic Gesture Searcher** (DYGS). Specifically, DYGS is an instrumentation of the core of the Java interpreter included in the JDK, which examines each bytecode to be interpreted before it is executed, as well as other indicators of the current state of the program, especially the execution stack.

5.6.1 Dynamic Gesture Candidate Selection with DYGS

The most basic approach that DYGS uses to finding gestures is similar to the approach used in §5.2.2, namely, by keeping track current gesture state. The chief difference between this analysis and static analysis is that the string of instructions we are searching through is now the sequence of instructions that are being executed, rather than the sequence of instructions obtained from the `.class` file layout. Therefore, the gesture state is updated as we examine each executed instruction. Using gesture state, DYGS maintains a distribution of the lengths of all gestures that have been encountered so far. Whenever it is determined that a gesture cannot continue beyond a certain point in the execution (we have reached a gesture-terminating state), the gesture is "rolled back" to the last significant instruction that was executed (*i.e.* a `getfield` or `putfield`), then added to the distribution according to its current length.

5.6.2 Finding Non-Sequential Gestures

Determining which instructions can and cannot continue a gesture is a somewhat complex process. If we simply look for sequentially executed instructions, as we did in §5.3, we can be sure that we will only find valid gestures, for reasons already explained, but we cannot guarantee that we will find all valid gestures, or even a boundable percentage of them. This

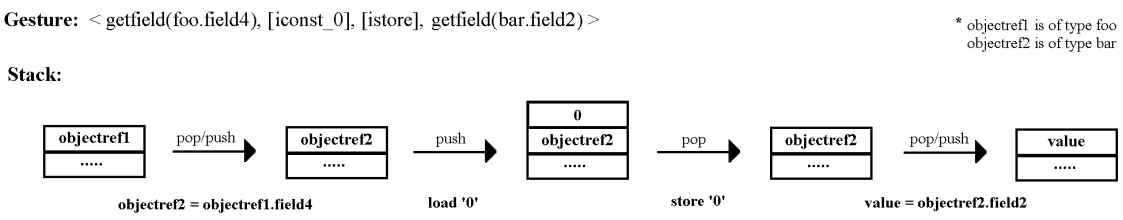

Figure 5.1: A Valid Non-Sequential Getfield Gesture

is because some potential gestures could be interleaved with other instructions that do not change the program's state from the gesture's perspective, as is shown in Figure 5.1.

Since a `getfield` chain only influences and is influenced by the item on top of the stack when each `getfield` is executed, it is conceivable that items could be added and removed from the stack between `getfield` instructions while the stack slot used by the `getfield` chain remains unchanged and valid. If this stack slot remains unchanged and valid, we can then preload the final result of the gesture in this slot with our memory macro where the first `getfield` would have been executed, with the assurance that it will still be on the stack where the last `getfield` would have been executed.

Therefore, any time the top of the stack contains the result of a previous `getfield` when a new `getfield` is executed, these two instructions could potentially be part of the same gesture. However, there is one important additional criterion that these non-consecutive gestures must also meet. The top of the stack that contains a `getfield` result also must not be examined or used by any other instructions before it is examined by the next `getfield`, to ensure atomicity of the `getfield` chain. If this was not the case, preloading the result of the entire chain would not be possible because another instruction would need access to the result of one of the intermediate steps (see Figure 5.2).

To account for these non-sequential gestures in the regular expressions we defined in §2.4, we can now define the four gesture types as:

1. $[g(g+n)^*g]$ *(Non-Sequential Getfield-Getfield Gestures)*
2. $[s(g+n)^*g]$ *(Non-Sequential Getstatic-Getfield Gestures)*
3. $[g(g+n)^*p]$ *(Non-Sequential Getfield-Putfield Gestures)*
4. $[s(g+n)^*p]$ *(Non-Sequential Getstatic-Putfield Gestures)*

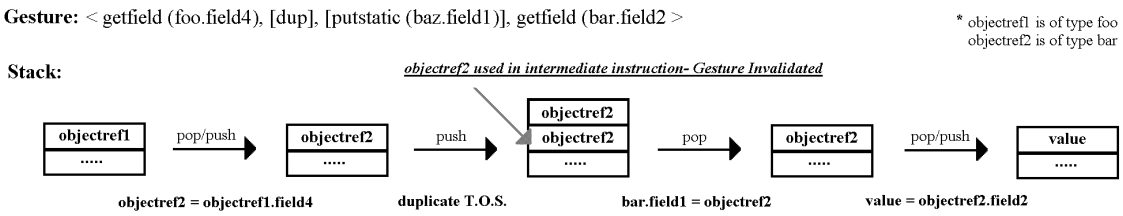

Figure 5.2: An Invalid Non-Sequential Getfield Gesture

or, simply $[(s+g)(g+n)^*(g+p)]$, where n represents any instruction that does not modify or access the stack in such a way that would render the gesture invalid.

To find non-sequential `getfield` chains, DYGS stores the address of the top of the stack and the item stored at that address (an object reference) after each `getfield` is executed. When a new `getfield` is encountered, the stored stack address and object reference are compared to the current address and item at the top of the stack. If the stored and current `getfield`s are consecutive, or if all interspersed instructions have no net effect on the stack, then the old address and object reference will be equal to the new address and object reference at the top of the stack, and the `getfield`s can be considered part of the same gesture. Otherwise, we know that the result of the previous `getfield` instruction has been overwritten with some other value (if the reference has changed), or the stack is now of a different size (if the top-of-stack address has changed), so the gesture candidate is rolled back to the previous `getfield` and recorded, while the current `getfield` is considered as part of a new gesture.

To ensure that no interspersed instructions examine any of the intermediate values in the gesture, DYGS contains a list of all JVM bytecode instructions which use the value on top of the stack. If the slot used by a `getfield` chain is currently on top of the stack, DYGS checks that each interspersed instruction executed by the program is not on this list of "gesture-ending" instructions. If an interspersed instruction is on the list, the current gesture candidate is rolled back and recorded at that point.

5.6.3 Dynamic Analysis of Getfield-Putfield Gestures

Finding `getfield-putfield` gestures is a process very similar to the one described above for finding `getfield` gestures. Finding sequential gestures is an almost identical process, where we increase our gesture candidate's size as we find each consecutive

`getfield`, and then record the candidate when we see any other instruction. The only difference is that when a `putfield` is encountered, it is included in the final gesture that is to be recorded, unlike any other non-`getfield` instruction type.

Finding non-sequential `putfield`-`getfield` gestures is also similar to the analog for `getfield` gestures. In this case, the gesture-ending `putfield` instruction could be separated from the last `getfield` instruction of a chain by some number of interspersed instructions that do not affect the stack. In addition to enforcing the standard constraints on interspersed instructions regarding the top of the stack, we would have to enforce the constraint that the penultimate item of the stack (which holds the address of the object where the result of the chain is stored) also remains constant for the entire length of the gesture. This constraint is required because we would need this stack item at the beginning of the gesture, where the macro would be executed, and at the end of the gesture, where the `putfield` is originally executed, in order to maintain the transparency of the gesture.

Instead of explicitly tracking this constraint, we can take advantage of the inherent structure of the stack to enforce it. Specifically, if we enforce the constraint that the top of the stack is never altered between each instruction in the gesture (as we did for `getfield` chains), the structure of the stack mandates that all items further down on the stack are not altered as well. Thus, we know that if the validity of the item on top of the stack is preserved between each step of the gesture, the penultimate item is also preserved, and our `getfield`-`putfield` chain is valid.

5.6.4 Additional Features of DYGS

In addition to determining the frequency of each type of gesture that occurs in a program's execution, DYGS was designed to provide several other types of information as well.

- **Gesture Printouts:** DYGS provides a command line argument that allows the user to tell DYGS to print out all gestures it encounters which meet a certain set of criteria. The criteria are used to narrow down the number of output gestures, since printing out all gestures would in some cases consume a number of lines on approximately the same order as the program's total executed instruction count. Specifically, the user can specify a minimum length of gestures to be printed out, as well as minimum and maximum instruction gap size between gesture instructions in the case of non-sequential gestures.

- **Instruction Timing:** In order to achieve a rough estimate of how much time could be saved by incorporation of detected gestures within memory macros, we would need to know how long instructions take to execute. Logically, different instruction types can take different lengths of time, so a simple average instruction time is not enough. To help gather information about specific gesture-related instructions such as `getfield` and `putfield`, DYGS includes an option that finds average execution times for a certain set of instruction types within the DYGS-modified JDK 1.1.8 interpreter. Due to the fact that this interpreter has been extensively modified and designed for flexibility, it is not nearly as optimized for speed as commercial-grade interpreters are, so the instruction timing numbers found here should not be taken as real-world values. However, the numbers can give us an idea of the time instructions take relative to the total execution time of the program and to each other. This in turn can give us a rough idea of what percentage of time could be saved by using memory macros (see §5.8).

5.7 Dynamic Analysis of C programs with SimpleScalar

To this point, our analysis has focused exclusively on Java programs. The referential transparency of the fields in Java classes, and the stack-based JVM (where we always know the exact location of relevant data items) lend themselves particularly well to gestures. However, it is worthwhile to investigate potential gestures in other languages as well, since a great number of programs today run on platforms very different from the JVM. In addition, since Java programs execute on a "Virtual Machine" which would in turn have its own set of memory-accessing patterns, it would make sense to find gestures in a lower-level language. Therefore, our next phase of experiments was designed to again answer the questions presented in §5.1, but on a different platform.

We chose to investigate C programs, based on this language's widespread use, and the numerous differences these programs have with Java programs. To do this, we needed a platform that could run C programs and was easily instrumentable, so that gesture-finding code could be added. The platform selected was **SimpleScalar** [12], a software-based system architecture that can emulate several common instruction sets, such as Alpha, PISA, ARM, and x86. Most C programs can be compiled into SimpleScalar assembly code, which can then be converted into object files and linked to form a SimpleScalar executable. These executables can in turn be run on a variety of simulator modules provided with the package. Here, the sim-profile module, a functional simulator written in C which provides various

program profiling information, was the target of the instrumentation. The resulting program will be referred to as **DYGS-SS**, Dynamic Gesture Searcher for SimpleScalar.

SimpleScalar is a register-based architecture, so memory load and store instructions operate on a group of registers, 32 in this case. When finding gestures in Java, we could investigate the top of the stack at any point in the program's execution to determine whether a gesture could continue past the current instruction or not. Because of the multiple registers in SimpleScalar, we can no longer simply look in one place. Instead, we need to be able to keep track of the gesture state of each register, which in this case refers to the maximum-length gesture that the value held in each register could be a part of. We refer to the process of keeping track of these values as **register shadowing**. We also need to be able to transfer gesture state from a source register to a destination register when a memory load instruction occurs. These factors make the investigation of gestures in this environment considerably more difficult by requiring additional memory and processing time, so in this set of experiments we constrained ourselves to relatively simple gesture-recognition methods.

5.7.1 Benchmarks in C

Obviously, a different set of benchmarks, written in C, was needed for analysis on this platform. We used the **CommBench** benchmarks [17], a set of applications designed for use in a network processor environment. We felt these repetitive, computationally-intensive programs would benefit significantly from use of memory macros, since they may be performing the same type of gesture many times. CommBench consists of eight programs, but not all were used in all experiments due to compatibility issues with the SimpleScalar platform. The individual programs that were used in the experiments are as follows:

CAST: An implementation of the CAST-128 block cipher algorithm, whose main computation consists of encryption arithmetic.

DRR: A deficit-round robin scheduling algorithm, whose main computation is maintenance of queues and resource tokens.

FRAG: An application that fragments IP packet headers. The main computation is recalculation of the IP header checksum for each fragmented header.

REED: An implementation of a Reed-Solomon Forward Error Correction algorithm, which performs redundancy coding and error correction on data that was Reed-Solomon encoded.

RTR: A routing lookup program based on the radix-tree routing algorithm, whose main computation is traversing routing trees and comparing address prefixes.

ZIP: An implementation of the Lempel-Ziv (LZ77) compression algorithm, which compresses and decompresses data using entropy encoding.

5.7.2 Dynamic Gesture Candidate Selection DYGS-SS

Much as we defined a language of valid gesture types in Java with `getfield`, `putfield`, and `getstatic` instructions, we need to define a set of valid gestures in SimpleScalar opcodes. First, let us consider what constitutes a gesture on a register-based machine. Ideally, we are looking for a sequence of instructions where each instruction accesses memory using the address put in a register by the previous instruction in the chain, like the one shown in §1.2.

In memory accessing instructions in SimpleScalar, the destination of the value retrieved from memory is not necessarily the same as the source of the memory address, as it was in Java, where both were always the top of the stack. Here, the source and destination are both specified as registers, which may or may not be the same. Therefore, through register shadowing, we must keep track of the longest gesture that the value in each register currently could be a part of. Then, if we encounter an memory-accessing instruction which uses that register as the source register, the gesture length associated with that register must be incremented and moved to the instruction's destination register. Conversely, when any other type of instruction accesses a register, the gesture could not continue beyond that point, so the register's current gesture length would be recorded then reset to 0.

Thus, our set of valid gestures can be most accurately defined as a chain of memory accessing (load) instructions[1], where each load instruction in the chain uses the previous load instruction's destination register as a source register, and no other instruction between those two load instructions accesses or modifies that register.

To accomplish this task in SimpleScalar, we created an array of **shadow registers**, one for each real register, which store the gesture length corresponding to the value in each real register. All register access and modification in SimpleScalar is done through C preprocessor macros, so modification of these macros allows us to do our accounting with these shadow registers on the fly, as instructions are processed. Gesture lengths are

[1]We use the generic term "load instruction" to refer to the entire set of memory load instructions in the SimpleScalar ISA (*e.g.* LoadByte, LoadWord, etc.). Please refer to the SimpleScalar documentation [12] for more information on specific instructions.

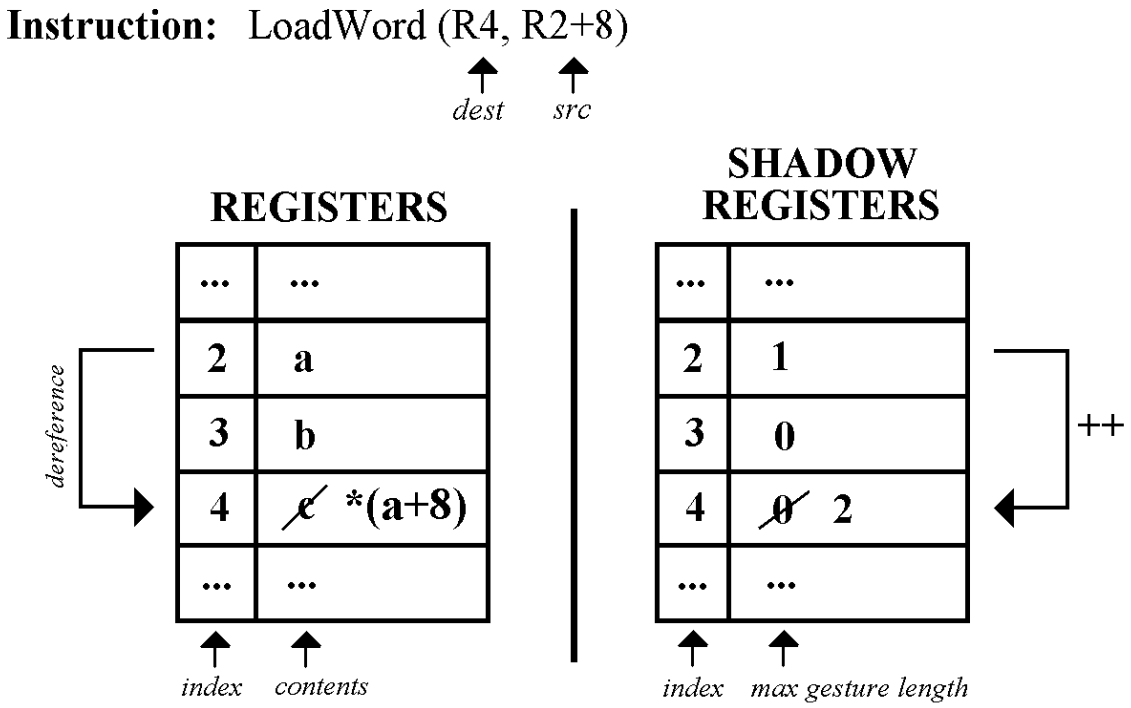

Figure 5.3: Example of a Shadow Register Manipulation

recorded as soon as we reach a point where they cannot continue, and cumulative gesture length distributions are calculated and then displayed when the program exits. An example of the shadow register manipulation for one step of a gesture is shown in Figure 5.3

5.7.3 Additional Features of DYGS-SS

In addition to printing out the distribution of gesture sizes as described above, there is also support in the DYGS-SS code to print the largest gesture encountered, and to determine whether or not gestures of the same length are identical.

To facilitate a more generalized form of gesture analysis and to provide a launching point for future work in this gesture recognition, DYGS-SS also contains the option to print out the frequencies of all consecutively-executed opcode 2-tuples encountered in the execution of the program.[2] Here, we are not looking for gestures of any particular type, but are interested in finding out the most frequently-occurring 2-tuples (representing potential

[2]Tilman Wolf presents a similar investigation of ordered instruction frequencies within the CommBench benchmarks in his doctoral dissertation [16].

gestures of length 2), in hopes that some of these tuples could be somehow encompassed in new gesture types. Essentially, we are approaching the problem from the opposite end, by identifying frequently occurring tuples that may be mapped to gestures, rather than identifying gestures that may occur frequently. At this point, this idea has not been fully investigated, and could be explored further if future work on this subject was carried out (see Chapter 6).

5.8 Memory Macro Simulation

The Memory Macro Simulation package, also known as MacroSimulator, was conceived as a front end to our gesture-finding programs to help solve several problems with the relative obtuseness of the gesture data that these programs present, and to better analyze the impact of memory macros within different environments. Simply looking at statistics on the number of gestures found and instruction execution times does not necessarily give a coherent picture of why memory macros would be useful. Specifically, it may be difficult for individuals to achieve a concrete understanding of how use of memory macros could result in time savings. To this end, MacroSimulator lets the user specify various high-level characteristics of the "machine" which is being simulated, including information about the average execution times of several instruction classes, such as `getfield`s, `putfield`s, or all instructions that do not access memory. Analysis of simulated program execution times under different sets of these characteristics may allow us to gain a better understanding of which environment and program types are most conducive to incorporation of memory macros.

MacroSimulator was designed for the purpose of increasing the understandability of the data, so emphasis was placed on simplicity in the interface. More accurate simulators could be crafted, and the methods used here to simulate are not necessarily the most precise, but they serve the purpose of making the performance gains more understandable with a minimal amount of processing effort. To this end, MacroSimulator currently only simulates memory macros in Java, because they are conceptually simplest, but there is no reason it could not be extended to support more complex gestures or gestures in other languages.

MacroSimulator can be decomposed into several components, each of which may or may not be included according to the needs of the user. Figure 5.4 shows the interaction between these components, which are defined as follows:

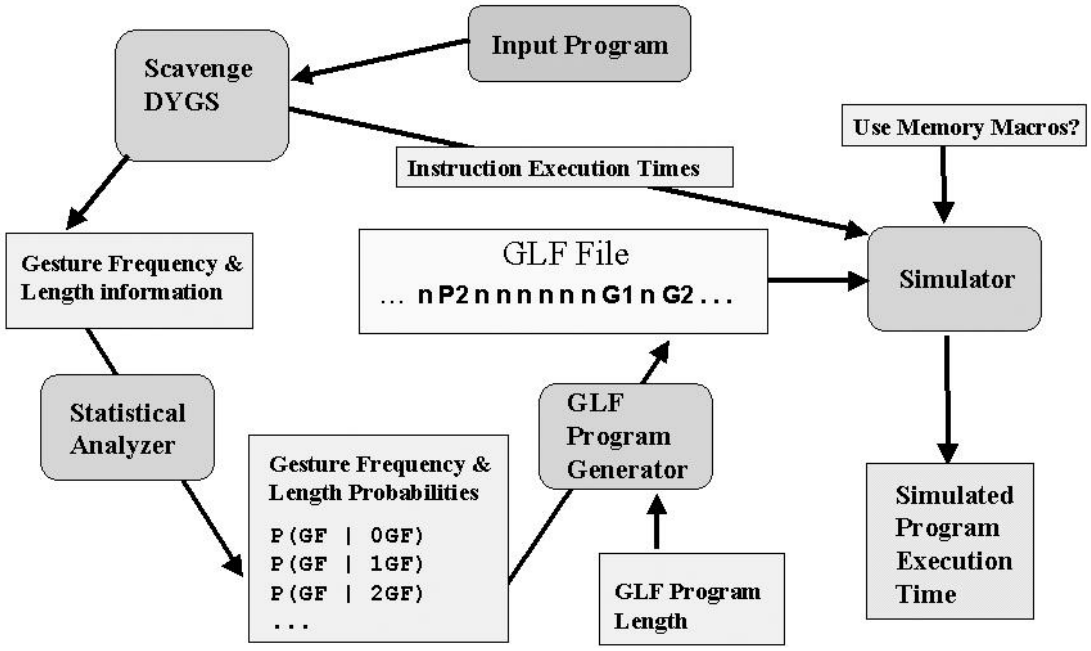

Figure 5.4: Interaction between MacroSimulator Components

5.8.1 Simulator

The core of the program, a Java application which takes in a file that describes the gestures used within a program and "simulates" execution of the program using a set of provided average instruction execution times. The Simulator has two main modes, one that uses the virtual CPU to handle each step of a multiple indirection, and one that allows the virtual memory module to handle the entire indirection chain in one atomic step (*i.e.* using memory macros). Comparing the results of the same program run under both these modes should give the user an idea of how much time could be saved by using memory macros. In order to simulate program execution times, the program needs access to individual execution times of instruction classes mentioned above. These values can be loaded with a command-line argument, or the Simulator can use a set of standard, hard-coded values taken from various test platforms. As the Simulator processes the input program, it passes through the states of the state model and calculates how much time must be added to the total program for each of those states, based on the provided instruction execution times.

The core simulator does not operate on the program to be analyzed, but rather an intermediate file which encapsulates the gesture information of that program. Specifically,

the program to be analyzed goes through two steps of processing before it is fed to the core simulator. These two steps are respectively handled by the following two components.

5.8.2 Statistical Analyzer

This component reduces the input program to an a gesture data file. This file contains the number of instructions in the file to be processed, and the probabilities that a `getfield` is the next instruction encountered during execution, for chains of varying lengths. For example, this file contains such information as:

P(`getfield` | last instruction was not a `getfield`)

P(`getfield` | instruction was a `getfield`)

P(`putfield` | instruction was a `getfield`)

P(`getfield` | last two instructions were `getfield`s)

..and so on

5.8.3 GLF Program Generator

This component handles generation of a variable-length program from a gesture data file. In this step of processing, we generate a new "program" of a user-specified length based on the probabilities given in the gesture data file. This "program", which is designed to run in the core simulator, is not composed of real instructions but rather a sequence of symbols indicating the length of the gesture that the corresponding current instruction would be a part of (including length 1 for instructions that are not part of a gesture), which we call gesture-length format (GLF). Since our simulator is only concerned with whether or not an instruction is part of a gesture, this is all the information we need to include in GLF. This choice means that simulation of instruction execution times are not always going to be accurate on a per-instruction basis, since all non-`getfield` gestures are grouped into the same category. However, the average non-`getfield` instruction execution time over the length of the entire GLF program, the figure which we are more concerned about when determining how effective memory macros are, should be fairly accurate.

We chose to gather probabilities and generate our own programs rather than simply making a direct analog of the input program in GLF for several reasons. First, this approach allows us to make programs similar to the input program but in radically different sizes. Secondly, this approach also gives us the ability to generate many programs that are similar to the input program in composition but still unique. Generating programs this way allows

us to find new test cases in a much more efficient way than searching for and individually analyzing true candidate programs.

5.8.4 Visualizer

It may be difficult for individuals to visualize the interplay between the CPU and memory which would cause a reduction in bus cycles if memory macros are used. To alleviate this problem, a GUI created by Chris Hill with the Java Swing package has been integrated with the rest of the MacroSimulator package. The GUI's menus provide access to a variety of options that are recognized by the Simulator, including commands to load one or more GLF files, run and reset the simulation, or run the program in both simulation modes (with and without memory macros) in parallel. The Visualizer also provides support for user customization of the instruction class timing parameters and other simulated program characteristics discussed above. The GUI also has the ability to visually represent the exchange of information between the CPU and memory units of a machine, to provide a clearer picture of the benefit of using memory macros.

5.8.5 Use of MacroSimulator

A user of MacroSimulator may choose to use only certain aspects of the entire package, which we hoped to facilitate by our separation of the various modules. For example, a user looking for increased execution speed may choose to not include the Visualizer, or could choose to generate a GLF file in a different way (perhaps through a direct mapping of a source program) and not include the GLF Generator and Statistical Analyzer. In addition, any gesture-finding programs that could be crafted to generate a data file in the same format as those created by Statistical Analyzer could interface with MacroSimulator. Chris Hill developed a program which gathers the same statistical data from Scavenge, and a program could just as easily be created to generate statistics from our javap experiments or any other gesture-finding approach.

5.9 Experimental Results

5.9.1 Javap

The results from our analysis with javap (see Figure 5.5) showed that double indirections occurred in about half the benchmarks, and there were no instances of triple indirections or

SPEC Benchmarks	
Name	**Gestures Found**
check	0
compress	0
jess	0
raytrace	3
db	0
javac	216
mpegaudio	102
mtrt	0
jack	14

Java API Packages	
Name	**Gestures Found**
java.io	4
java.lang	0
java.math	0
java.net	0
java.security	0
java.text	44
java.util	3

Figure 5.5: Length-2 Gestures Found by Javap

larger anywhere within the classes that were tested. Only two benchmarks had frequently occurring successive gestures within the code: javac (216) and mpegaudio (102). No other benchmark had a gesture count above 14, and no chains were found in six of the benchmarks. Among Java API `.class` files, java.text had the most, with 44 `getfield` chains among its classes, but no other packages had above four, and four packages had none.

5.9.2 Scavenge

The number of gestures found by Scavenge in each of the benchmarks (see Figure 5.6) significantly differs from the number found for each benchmark in the javap experiments (see Figure 5.7). In this experiment, we see a more even distribution of gesture counts among all benchmarks. Some benchmarks that had very few or no gestures in javap have many in this experiment, while others saw their gesture count radically decreased. These results are not in conflict with our claim that Scavenge has more accurate and powerful gesture recognition capabilities than javap. Differences in the number of gestures found

Benchmark Name	Gestures Found
check	47
compress	3
jess	26
raytrace	6
db	3
javac	162
mpegaudio	0
mtrt	6
jack	117

Figure 5.6: Length-2 Gestures Found by Scavenge

are likely due to the fact that we are examining *all* the `.class` files used by a benchmark and *only* those files, whereas javap was limited to what was in the program's directory. For example, _200_check's gestures were found in parts of the Java API that the program used but were not in the program's directory, whereas _222_mpegaudio's decrease was due to the fact that it had `.class` files in its directory structure that were not referenced from any point within the program.

Our memory macro counting results (see Figure 5.8) show that the number of unique memory macros for a given benchmark was reduced by the field reordering algorithm in most cases, usually by 25-50% (see Figure 5.9). In addition, it seems that we are more successful in eliminating macros from those benchmarks that would need a large number of macros to begin with. This is what we would expect based on our heuristic, assuming the field numbers used in the macros are semi-random to begin with. Furthermore, we see that the heuristic used never increased the number of macros needed.

5.9.3 DYGS

As can be seen in Figure 5.12 and Figure 5.13, the number of gestures found by DYGS varied widely between benchmarks. Within each benchmark, the number found scaled roughly with the total number of instructions executed, perhaps indicating that the gestures encountered occurred in the heart of the program's computation and not only during initialization or other do-once tasks. Two benchmarks, _213_javac and _222_mpegaudio, contained gestures of a length greater than two, and none contained gestures of a length greater than three.

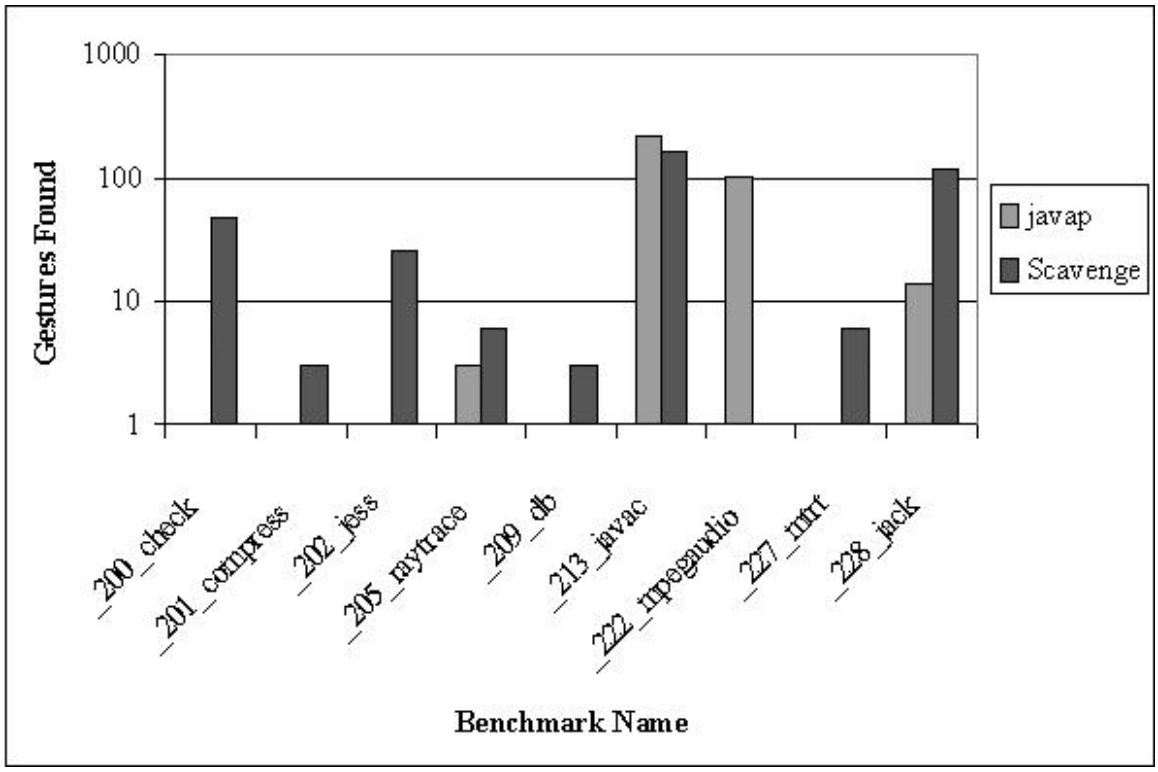

Figure 5.7: Gestures Found by Javap vs. Gestures Found by Scavenge

Benchmark Name	Macros Needed	
	Before Reordering	After Reordering
check	13	8
compress	2	2
jess	6	3
raytrace	4	3
db	2	2
javac	19	9
mpegaudio	21	16
mtrt	3	3
jack	6	2

Figure 5.8: Number of Macros Needed before and after Field Reordering

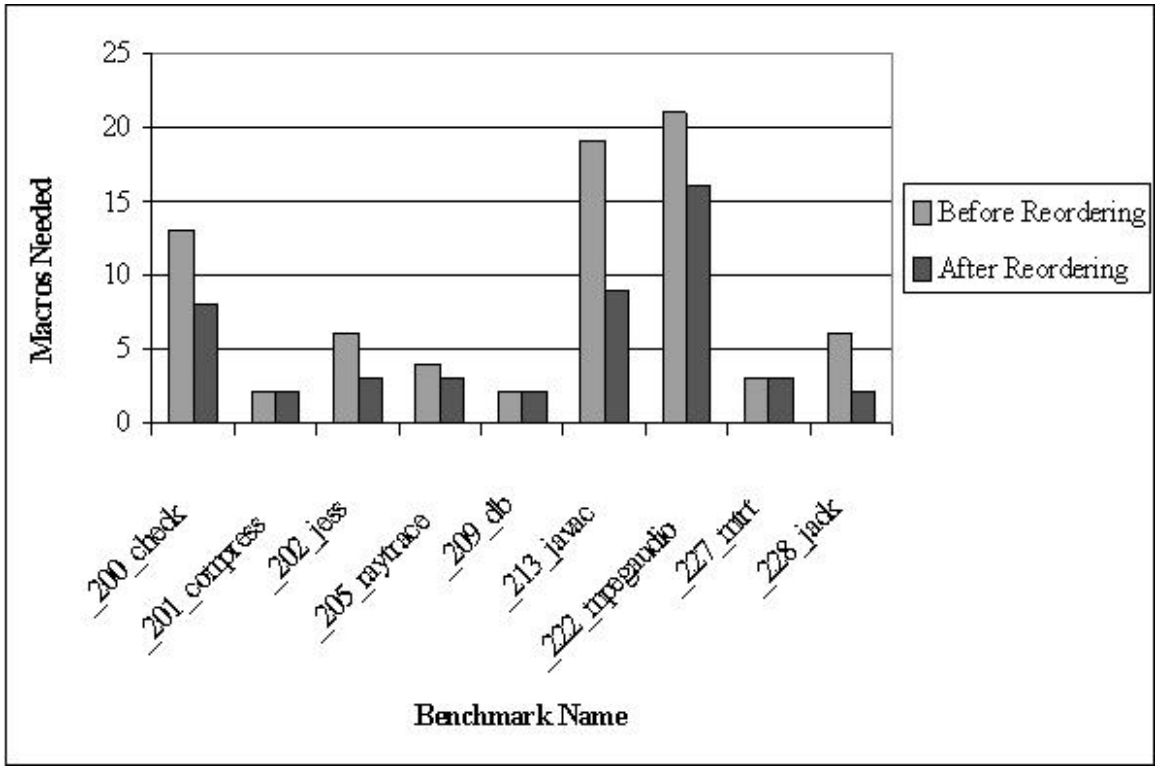

Figure 5.9: Macro Reduction through Field Reordering with Scavenge

The benchmarks with the longest gestures also tended to have the highest frequency of gestures. This high-frequency group contained the _202_jess, _228_jack, and _209_db programs, in addition to the two already mentioned. Even within the high frequency group, the overall frequency of gestures in proportion to the number of instructions executed was still quite low, below 1% of the all instructions. However, because these benchmarks are computationally intensive, the gesture count itself was still very high in the case of some of the larger benchmarks, occasionally reaching past seven decimal places.

In most benchmarks, `getfield-putfield` gestures occurred with a higher frequency than `getfield` gestures, the only exceptions being _209_db and _222_mpegaudio (see Figure 5.14, Figure 5.10, and Figure 5.11)[3]. In addition, the only length-3 gestures found in any benchmark were `getfield-putfield` gestures. No gestures involving `getstatic` were found in any of the benchmarks.

[3]Some size-100 benchmarks required more memory than was supported by the test platform and did not finish execution.

Benchmark		Instructions	Length-2
Name	Size	Examined	Gestures Found
check	1/10/100	20273308	4
compress	1	1061896761	7
compress	10	1280284405	13
compress	100	2364623359	79
jess	1	9922216	1574
jess	10	137656198	665
jess	100	1252658299	46499
raytrace	1	60305684	10
raytrace	10	181155032	10
db	1	2839136	1089
db	10	96462412	102723
db	100	4509395032	23126420
javac	1	7996005	295
javac	10	73909249	43455
javac	100	1513149003	1089496
mpegaudio	1	150360205	190431
mpegaudio	10	1582552168	2165268
mpegaudio	100	14834492570	18948300
mtrt	1	58945118	10
mtrt	10	224181327	16
jack	1	178505154	33526
jack	10	355883979	67048
jack	100	1728250510	322600

Figure 5.10: Getfield Gestures Found by DYGS in SPECjvm98 Benchmarks

5.9.4 DYGS-SS

In our examination of gestures in the CommBench Benchmarks, all contained gesture lengths of at least three and two contained gestures of length four or longer. In addition, some benchmarks contained very high percentages of instructions that could be included in gestures. In both the REED encoder and RTR, potential gestures encompassed over 10% of a program's total instructions, whereas no more than 1% of any Java benchmark's total instructions were encompassed by potential gestures (see Figure 5.15 and Figure 5.16).

One particularly interesting benchmark was the ZIP decoder, which did not contain an unusually high percentage of gestures of any length, but did contain several extraordinarily long gestures. Whereas no other benchmarks had gestures of a length greater than 4,

Benchmark		Instructions	Number of Gestures Found	
Name	Size	Examined	Length-2	Length-3
check	1/10/100	20273308	35	0
compress	1	1061896761	58	0
compress	10	1280284405	64	0
compress	100	2364623359	314	0
jess	1	9922216	34199	0
jess	10	137656198	97941	0
jess	100	1252658299	3647386	0
raytrace	1	60305684	69	0
raytrace	10	181155032	78	0
db	1	2839136	44	0
db	10	96462412	44	0
db	100	4509395032	44	0
javac	1	7996005	1481	86
javac	10	73909249	249183	100816
javac	100	1513149003	7638809	3377757
mpegaudio	1	150360205	1214	312
mpegaudio	10	1582552168	11674	3576
mpegaudio	100	14834492570	93169	31244
mtrt	1	58945118	37	0
mtrt	10	224181327	40	0
jack	1	178505154	33981	0
jack	10	355883979	67928	0
jack	100	1728250510	330841	0

Figure 5.11: Putfield Gestures Found by DYGS in SPECjvm98 Benchmarks

this benchmark was found to have several long gesture types, up to a length of 128 instructions. The full distribution of instruction lengths can be seen in Figure 5.17. These long gestures occurred in the process of Huffman decoding a block of data. Specifically, the zip program performs decoding using a multi-level table lookup, and maintains a linked list of all the dynamic Huffman tables it creates for each block of data. When a Huffman table is no longer needed, it is freed, which requires a traversal through the linked list to find the table in question. This traversal accounts for the long gestures seen in the results.

Benchmark		Instructions	Gestures Found		Bus Cycles
Name	Size	Examined	Length-2	Length-3	Saved
check	1/10/100	20273308	39	0	39
compress	1	1061896761	65	0	65
compress	10	1280284405	77	0	77
compress	100	2364623359	393	0	393
jess	1	9922216	35773	0	35773
jess	10	137656198	98606	0	98606
jess	100	1252658299	3693885	0	3693885
raytrace	1	60305684	79	0	79
raytrace	10	181155032	88	0	88
db	1	2839136	1133	0	1133
db	10	96462412	102767	0	102767
db	100	4509395032	23126464	0	23126464
javac	1	7996005	1776	86	1948
javac	10	73909249	292602	100816	494234
javac	100	1513149003	8728305	3377757	15483019
mpegaudio	1	150360205	191645	312	192269
mpegaudio	10	1582552168	2176942	3576	2184094
mpegaudio	100	14834492570	19041469	31244	19103957
mtrt	1	58945118	47	0	47
mtrt	10	224181327	56	0	56
jack	1	178505154	67507	0	67507
jack	10	355883979	134976	0	134976
jack	100	1728250510	653441	0	653441

Figure 5.12: Total Gestures Found by DYGS in SPECjvm98 Benchmarks

5.9.5 MacroSimulator

Various datasets can be generated with MacroSimulator by adjusting its source data and simulation parameters. Here we simply give an example of the data that can be generated from this package, to show how MacroSimulator can be used to help us understand how various factors affect program execution time.

Suppose we wish to see what percentage of execution time could be saved in a 100,000-line Java program where 10% of instructions are length-two `getfield` gestures and 1% of instructions are length-3 `getfield` gesture. We can enter these percentages into a statistics file, which can then be loaded into the GLF Program Generator to generate a 10,000-element GLF file (or any number of files) with those approximate characteristics.

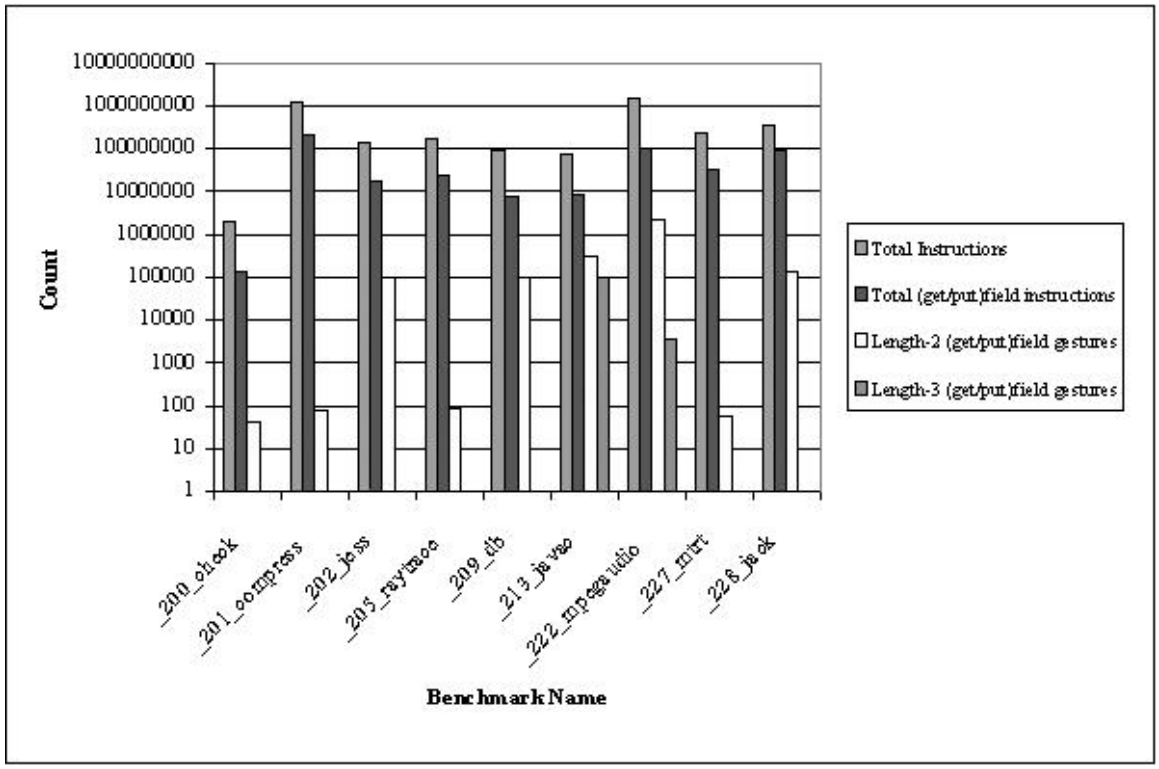

Figure 5.13: Number of Gestures Found in Size-10 SPECjvm98 Benchmarks with DYGS, by Length

This GLF file can then be loaded into the Simulator. Here, we choose not to use the default instruction execution times, so we specify to the Simulator that `getfield` instructions take on average 15 ns, all other instructions take 20 ns on average, and the processing of a macro within PIM takes an additional 2 ns over standard memory retrieval time. The Simulator, running once in each of its two modes, gives us an execution time of about 114,000 ns using memory macros and 116,000 ns not using memory macros.

Since the process of finding the necessary macros and loading them into memory can be done statically, the processing time of this step is not included in the simulation. However, we can use the results from our simulations to determine how fast we would have to do this macro finding in order to achieve an overall performance gain. In our above example, if the program is only run once, then the macro processing phase must take less than 2000 ns in order to achieve an overall performance gain. Likewise, if the program is run twice, then macro processing must take less than (2*2000 ns) = 4000 ns, and so on. An

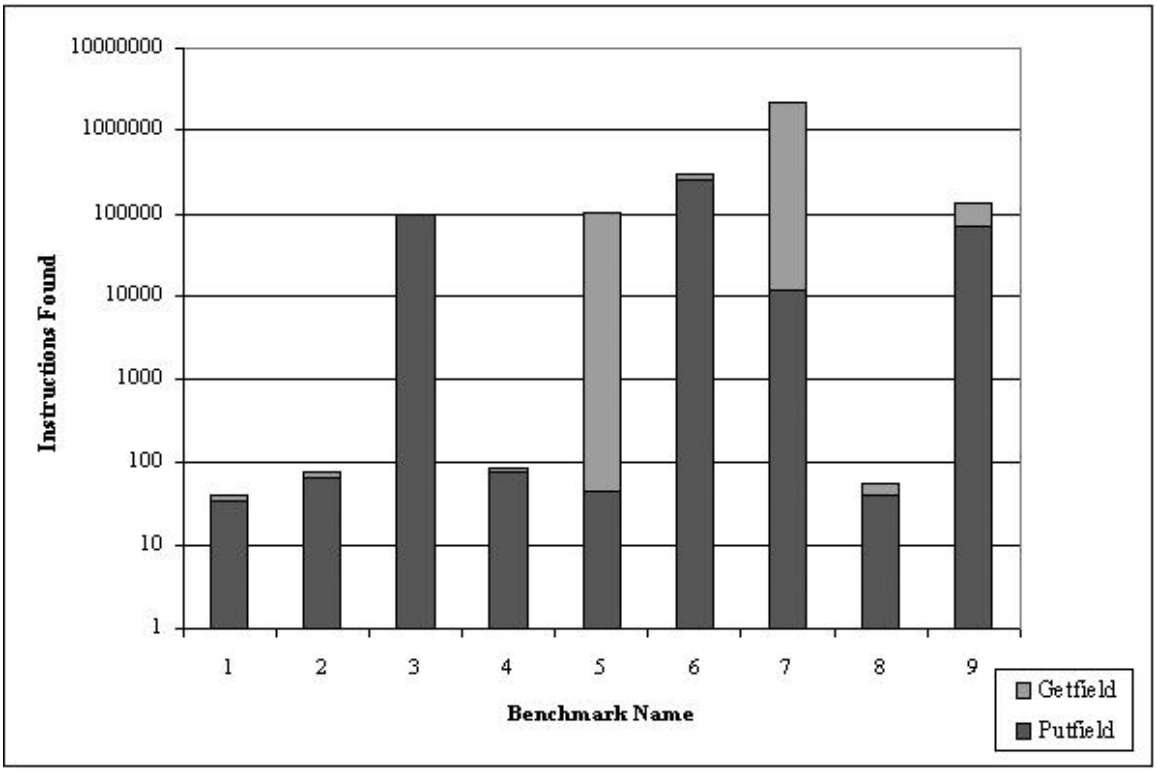

Figure 5.14: Number of `getfields` vs. Number of `putfields` in Length-Two Gestures, for Size 10 Benchmarks

Benchmark Name	Instructions Examined	Number of Gestures Found			Bus Cycles Saved
		Length-2	Length-3	Length-4	
cast	138679064	27889219	384508	0	28658235
drr	213072619	40303573	32033378	0	104370329
frag	43441982	6616877	1109	0	6619095
reed (encoder)	622925521	100709785	19417	0	100748619
reed (decoder)	1205431411	146813094	19422	0	146851938
rtr	1100034475	305671607	67502675	3630787	74764249
zip (encoder)	227631430	46360791	211	0	46361213
zip (decoder)	39909399	8506706	249	21	8511166

Figure 5.15: Gestures Found by DYGS-SS in CommBench Benchmarks

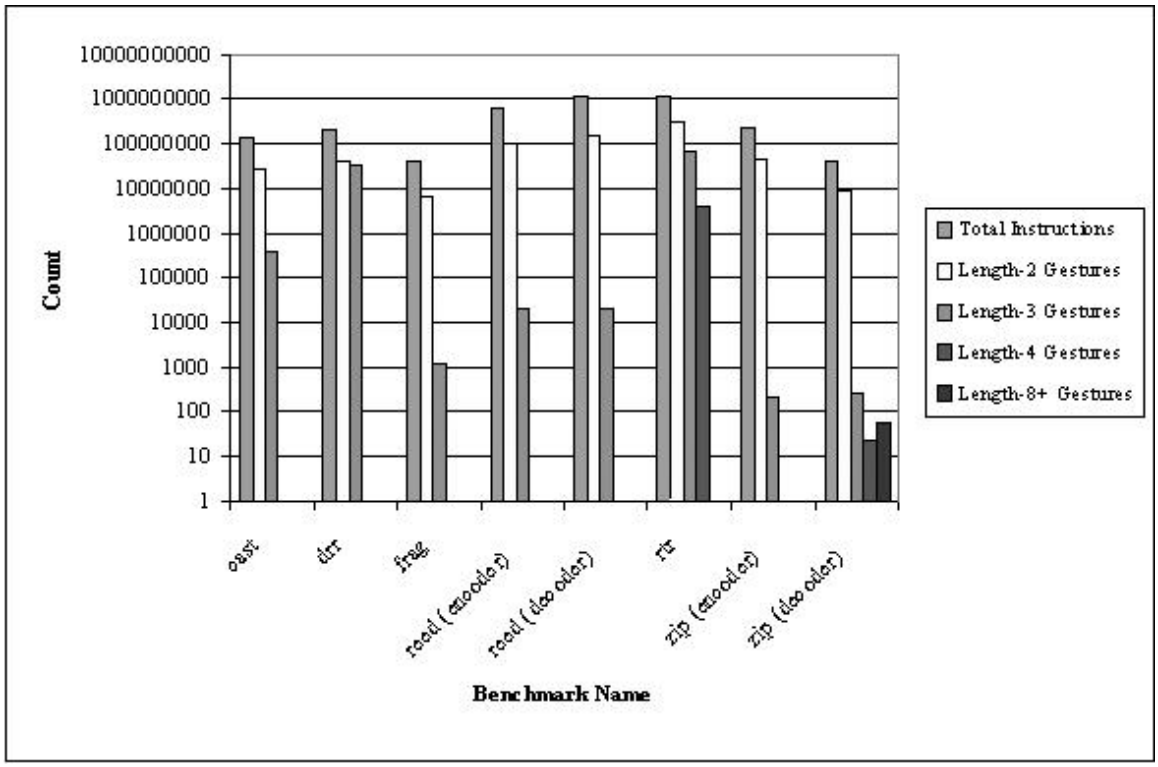

Figure 5.16: Number of Gestures Found in CommBench Benchmarks with DYGS-SS, by Length

example set of macro processing times calculated with this method for each of SPECjvm98 benchmarks is listed in Figure 5.18.

It should be clear from this example how MacroSimulator can be used to investigate the effects of many other hardware and software properties, such as:

- Gesture Frequency
- Gesture Length
- Program Length
- Program Memory-Intensiveness
- `getfield`/`putfield` Instruction Execution Times
- Disparity between `getfield`/`putfield` Instruction and Non-Gesture Instruction Execution Times

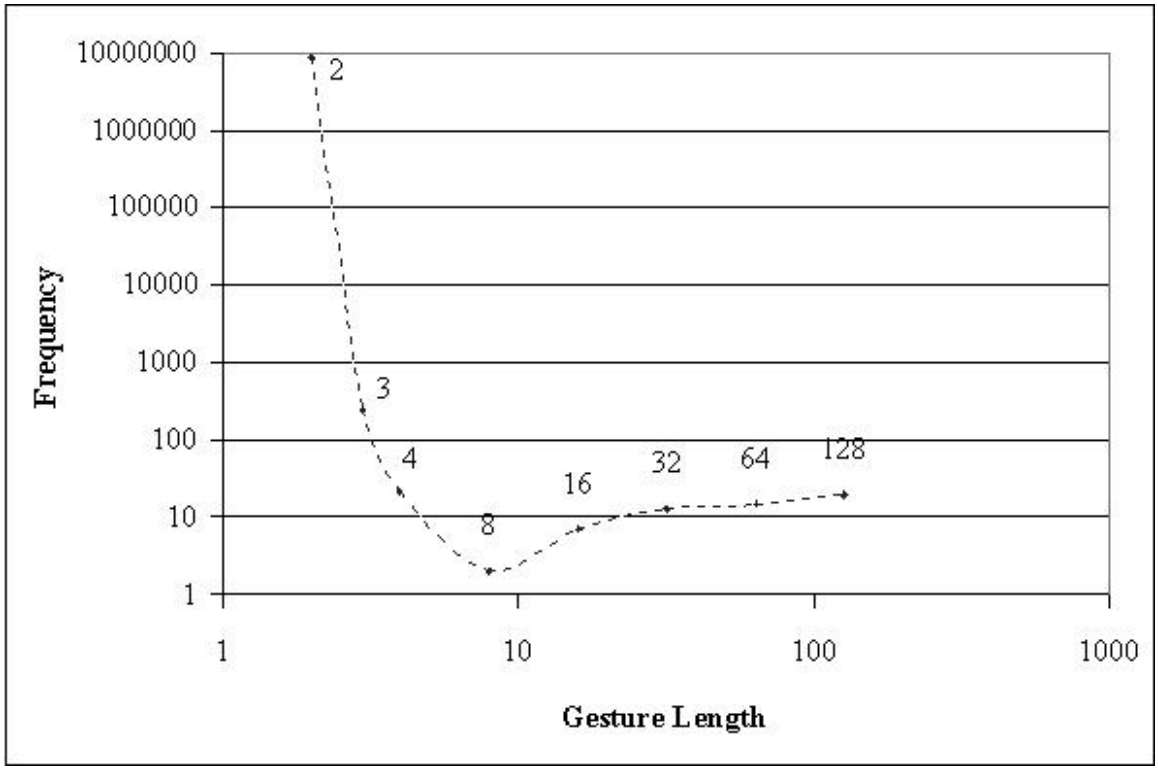

Figure 5.17: Gesture Distribution by Length in Zip Decoder Benchmark

- On-PIM Gesture Execution Times

on the execution time savings of memory macro use.

5.10 Conclusions

5.10.1 Javap

Our results from the javap analysis show that `getfield` gestures do occur in the bytecode of certain Java `.class` files. In some cases, these gestures occur in many places within the code, although the gestures are limited to a length of two consecutive instructions. However, this approach does not give us complete information regarding how many gestures occur in *all* the `.class` files used in a given program, since we are limited to scanning a certain subdirectory structure.

Benchmark		Simulated Execution Time (sec)		
Name	**Size**	**with Macros**	**without Macros**	**Improvement**
jess	1	0.1787	0.1646	0.0141
jess	10	2.3605	2.3237	0.0368
jess	100	21.8655	20.9771	0.8884
db	1	0.0471	0.0446	0.0025
db	10	1.6514	1.6268	0.0264
db	100	140.0346	137.4501	2.5845
javac	1	0.1906	0.1802	0.014
javac	10	1.6690	1.4767	0.1923
javac	100	33.2832	28.9092	4.3740
mpegaudio	1	1.8902	1.8456	0.0446
mpegaudio	10	18.4694	18.1617	0.3077
mpegaudio	100	182.4440	179.1127	3.3313
jack	1	3.4778	2.3487	1.1291
jack	10	3.4848	2.3468	1.138
jack	100	64.2033	45.2139	18.9894

Figure 5.18: Simulated Execution Times of SPECjvm98 Benchmarks with and without Memory Macros

5.10.2 Scavenge

Using Scavenge, we were able to accurately determine the number of gestures that occur in the code of all `.class` files associated with a given program. These results reinforce the conclusion that we drew in §5.10.1, namely, that gestures do occur in the context of Java `.class` files, and can be recognized through static analysis methods. In comparison with javap, gestures were found in a higher percentage of the benchmarks examined, although the number of gestures found per program did not increases significantly, and the maximum gesture length was still two. In addition, we were able to determine that our field reordering heuristic often significantly reduces the number of macros that would be needed.

5.10.3 DYGS

Our analysis shows that less than 1 percent of the instructions executed in the SPECjvm98 Java programs can be composed into valid gestures, and that the percentage of gesture-compatible instructions varied widely between individual benchmarks. Although this percentage seems relatively small, the absolute number of potential gestures found was often

quite high. Most benchmarks contained over one thousand executable gestures and some executed over one million. We can then say that our results show that gestures may not occur with a high frequency in running Java programs, but may occur in relatively large numbers. Dynamic analysis also yielded gestures of length 3, which were not found by static analysis methods, indicating that some gestures do cross method boundaries.

5.10.4 DYGS-SS

The results from our dynamic analysis of the CommBench C programs are considerably more encouraging than our results from dynamic analysis of the SPECjvm98 Java programs. We found that memory-accessing gestures occur with greater frequency and in greater lengths in this set of benchmarks than in the SPECjvm98 set of benchmarks. These results indicate that use of register-indexed gesture macros could be as beneficial as the use of stack-indexed gesture macros, or perhaps even more beneficial.

Chapter 6

Future Work

There are many areas of research where memory-accessing gestures could be further investigated. In this chapter we briefly discuss several of the most interesting options.

6.1 Dynamic Macro Generation

In Chapter 5, we used dynamic analysis as a means of finding more accurate gesture frequency statistics, and claimed that only static analysis was well-suited for determining the set of macros needed. However, since most gestures occur many times in the course of a program's execution, it may be possible to identify gestures the first time they occur and then create a macro that could be used for each additional occurrence of the gesture in the program. This approach would allow us to realize the benefits of dynamic gesture searching while only slightly reducing the number of times each macro would be used.

6.2 New Gesture Types

We could also extend our work by searching for new gestures. In our experiments we confined ourselves to rigidly-defined gestures that were manually chosen for their simple, logical structure. However, it may be possible to design an automated process to find additional candidates. This automated process could potentially find gestures that are more complex and less obvious than those used in these experiments, which could result in greater performance gains through use of memory macros. Carrying this one step further, this process could be carried out dynamically, as has been discussed in §6.1.

6.3 Cache Integration

We have greatly simplified our discussion of gestures by ignoring the impact of cache. If macro processing was done only within main memory, caching would obviously lessen any performance gains that would be achieved through use of memory macros, since some percentage of gestures could be serviced entirely within cache. However, there is no reason to think that gesture-handling modules could not be used in multiple levels of the memory hierarchy. Each of these modules could work independently to service any gestures that reference memory entirely resident on their level, and pass any gestures that could be serviced on to the next level of memory. More ambitiously, a single integrated gesture-handling module could preside over all levels of the memory hierarchy, where the address accessed on each step of a gesture could be individually fetched from whatever level it resides at. This module could use bus snooping or some other technique to determine where each address of the gesture is located, then access it directly.

6.4 Implementation

Finally, the most obvious next step in this research is to implement a system that recognizes and processes gestures within memory. We have already demonstrated ways to determine the set of macros needed through static analysis, but as of yet we have not addressed the specifics of how these macros would be sent to memory before execution, or how they would be used during execution. These tasks would be handled by the PIM, which has been treated as a theoretical device for simplicity's sake in this experiment. Determining the specific architecture of the PIM gesture-handling module is therefore a natural extension of the present work. A hardware simulation using VHDL or some other hardware description language would be a logical way of accomplishing this task. The macro processing module could also be implemented in hardware, perhaps through use of a **Field Programmable Gate Array** (FPGA).

References

[1] S. Donahue, M. Hampton, R. Cytron, M. Franklin, and K. Kavi. Hardware support for fast and bounded-time storage allocation. In *Second Annual Workshop on Memory Performance Issues (WMPI 2002)*, 2002.

[2] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

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

[4] Lucas M. Fox, Christopher R. Hill, and Ron K. Cytron. Optimization of storage-referencing gestures. *Proceedings of CODES 2003*, 2003. Submitted.

[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] John Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, New York, NY, third edition, 2002.

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

[11] 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.

[12] SimpleScalar LLC. SimpleScalar. `www.simplescalar.com`, 2001.

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

[14] Sun Microsystems. Java Development Kit 1.1.x. `java.sun.com/products/jdk/1.1/`, 1998.

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

[16] Tilman Wolf. *Design and Performance of a Scalable High-Performance Programmable Router*. PhD thesis, Department of Computer Science, Washington University, St. Louis, 2002.

[17] Tilman Wolf and Mark A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proceedings of IEEE International Symposium on Performance Analysis*, pages 154–162, Austin, TX, April 2000.

Vita

Lucas M. Fox

Date of Birth	July 10, 1980
Place of Birth	Menomonee Falls, Wisconsin
Degrees	B.S. Summa Cum Laude, Applied Science, May 2003, from Washington University in St. Louis.

May 16, 2003