Report Number: WUCSE-2003-81

2003-10-03

# Optimization of Storage-Referencing Gestures

Lucas M. Fox, Christopher R. Hill, Ron K. Cytron, and Krishna Kavi

We describe techniques for identifying and optimizing memory-accessing instruction sequences. We capture a sequence of such instructions, with the goal of sending the sequence as a single instruction from the CPU to a smart memory subsystem (IRAM or PIM). With a software/hardware codesign, the memory-accessing gestures can be rewritten as succinct superoperator instructions, and the gestures themselves could vary at runtime. As a result, the CPU executes fewer instructions and the CPU-memory bus is charged less often, resulting in lower power consumption. Reduction in power can be crucial for constrained, embedded systems. We discover gestures using a static and... **Read complete abstract on page 2.**

# Optimization of Storage-Referencing Gestures

Lucas M. Fox, Christopher R. Hill, Ron K. Cytron, and Krishna Kavi

Complete Abstract:

We describe techniques for identifying and optimizing memory-accessing instruction sequences. We capture a sequence of such instructions, with the goal of sending the sequence as a single instruction from the CPU to a smart memory subsystem (IRAM or PIM). With a software/hardware codesign, the memory-accessing gestures can be rewritten as succinct superoperator instructions, and the gestures themselves could vary at runtime. As a result, the CPU executes fewer instructions and the CPU-memory bus is charged less often, resulting in lower power consumption. Reduction in power can be crucial for constrained, embedded systems. We discover gestures using a static and a dynamic approach, and we present data showing the presence of such gestures in real benchmarks (Java and C). We have shown the gesture-minimization problem to be NP-Complete, so we offer in this paper a heuristic approach the effectiveness of which we evaluate with experiments.

Department of Computer Science & Engineering

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

2003-81

# Optimization of Storage-Referencing Gestures

Authors: Fox, Lucas M.; Hill, Christopher R.; Cytron, Ron K.; Kavi, Krishna

October 3, 2003

Abstract: We describe techniques for identifying and optimizing memory-accessing instruction sequences. We capture a sequence of such instructions, with the goal of sending the sequence as a single instruction from the CPU to a smart memory subsystem (IRAM or PIM). With a software/hardware codesign, the memory-accessing gestures can be rewritten as succinct superoperator instructions, and the gestures themselves could vary at runtime. As a result, the CPU executes fewer instructions and the CPU-memory bus is charged less often, resulting in lower power consumption. Reduction in power can be crucial for constrained, embedded systems. We discover gestures using a static and a dynamic approach, and we present data showing the presence of such gestures in real benchmarks (Java and C). We have shown the gesture-minimization problem to be NP-Complete, so we offer in this paper a heuristic approach the effectiveness of which we evaluate with experiments.

# Optimization of Storage-Referencing Gestures*

Lucas M. Fox, Christopher R. Hill, and Ron K. Cytron
Washington University
Contact: cytron@acm.org

Krishna Kavi
University of North Texas

October 3, 2003

**Abstract**

We describe techniques for identifying and optimizing memory-accessing instruction sequences. We capture a sequence of such instructions, with the goal of sending the sequence as a single instruction from the CPU to a smart memory subsystem (IRAM or PIM). With a software/hardware codesign, the memory-accessing gestures can be rewritten as succinct superoperator instructions, and the gestures themselves could vary at runtime. As a result, the CPU executes fewer instructions and the CPU-memory bus is charged less often, resulting in lower power consumption. Reduction in power can be crucial for constrained, embedded systems. We discover gestures using a static and a dynamic approach, and we present data showing the presence of such gestures in real benchmarks (Java and C). We have shown the gesture-minimization problem to be NP-Complete, so we offer in this paper a heuristic approach the effectiveness of which we evaluate with experiments.

## 1 Introduction

Modern programming languages coupled with advances in software engineering have resulted in better code reuse, with the result that most data structures are aggregations of other data structures. While software data-structure reuse is to be encouraged, programs that contain a high level of such reuse can suffer from the "containering" problem. For example, Figure 1 shows the navigation through a set of structures that is required to obtain the color of a pixel. An initial reference to an image is required, but from there, the remaining steps of the navigation are prescribed by the structure layouts of the various data types.

In terms of activity on a computing platform, the navigation in Figure 1 requires fetching an address present at a location in memory and adding an offset to that location, repeating that process until the destination is reached. Each offset in the navigation is a constant, related to the distance from the start of a data structure to where the data of interest is stored. The code corresponding to the navigation in Figure 1 is

```
*(*(*(p+8) + 12) + 4)
```

We call such a pattern of indirect fetches and offset additions a *gesture*, and we describe how to optimize the performance of storage-referencing gestures through use of static and dynamic program analysis coupled with customized, programmable hardware.

Our central idea is to recognize gestures and then replace them with a single "macro" instruction that is sent to the storage system (memory) for execution. The CPU is free to perform other actions until the gesture has been resolved and the destination value has been fetched or stored. This scheme can work as follows:

- A set of useful gestures is preloaded into the storage subsystem; each gesture is given a unique number.
- A gesture is executed by sending an instruction to the storage subsystem containing the number of the gesture and a starting address. Additional bits may be used if the final operation is a `store` instruction to provide the new value.

* Sponsored by NSF under grant ITR–0081214

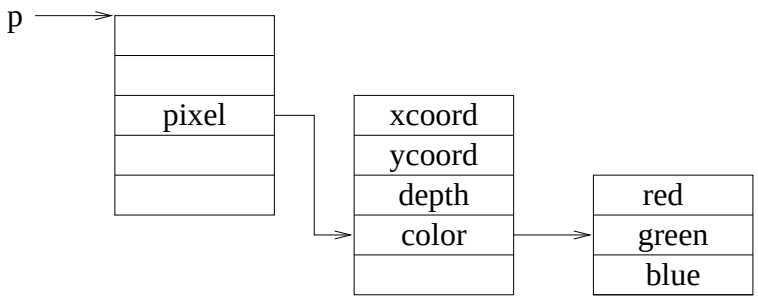


Figure 1: Navigation through data structures.

Without gesture substitution, the CPU and memory interact at each fetch within a gesture. The CPU–memory bus must be charged each time, only to send a value to the CPU so a constant may be added and then sent back to the storage subsystem. With the advent of Intelligent RAMs (IRAMs) and Processor In Memorys (PIMs), it is easy to deploy the addition circuitry in the storage subsystem to store and execute the gestures.

Some advantages of our approach are as follows:

- Computations reduce CPU power consumption, as the CPU–memory bus is charged less often.
- CPU throughput is improved, as address computations are offloaded to the storage subsystem.
- Fewer instructions are fetched by the CPU, as each gesture is coded as a single instruction.

In this paper, we present results from analyzing Java[1] and C programs for the presence of gestures. We describe an optimization problem for minimizing the number of gestures, and we present results from such optimization.

## 2 Dynamic Analysis

The benefit of using memory macros is directly related to the frequency, number and length of gestures that occur within a program.

- The savings due to a particular gesture increases the more frequently that gesture is executed.
- The demands on the storage subsystem for storing and representing gestures increase as more, unique gestures are found.
- The longer a gesture, the greater the savings in cycles, time, and power.

We investigate the above properties using *dynamic* analysis—statistics gathered from actually executing programs and observing storage-referencing gestures. While the gestures we find in this manner do not necessarily occur over all executions, the dynamic analysis provides an upper bound on the benefit of our approach on real benchmarks. Section 4 describes the success of our approach in actually finding and realizing storage-referencing gestures.

To accomplish the task of dynamically analyzing Java programs, we instrumented the Java Development Kit (JDK) 1.1.8 [11] 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 prior to its execution. Other components of the current state of a program are also included, most notably the execution stack.

Essentially, DYGS keeps track of the current *gesture state* as a program runs. A gesture state indicates whether the current instruction could be part of a gesture; if so, then the gesture's type and size are recorded. The state is then updated as each instruction executes. At each executing instruction, DYGS maintains a distribution of the lengths of all gestures that have been encountered so far. If a gesture in progress cannot continue beyond a certain point, then we say we have reached a gesture-terminating state. The gesture is "rolled back" to the last significant instruction that was

[1] Java is a registered trademark of Sun Microsystems, Inc.

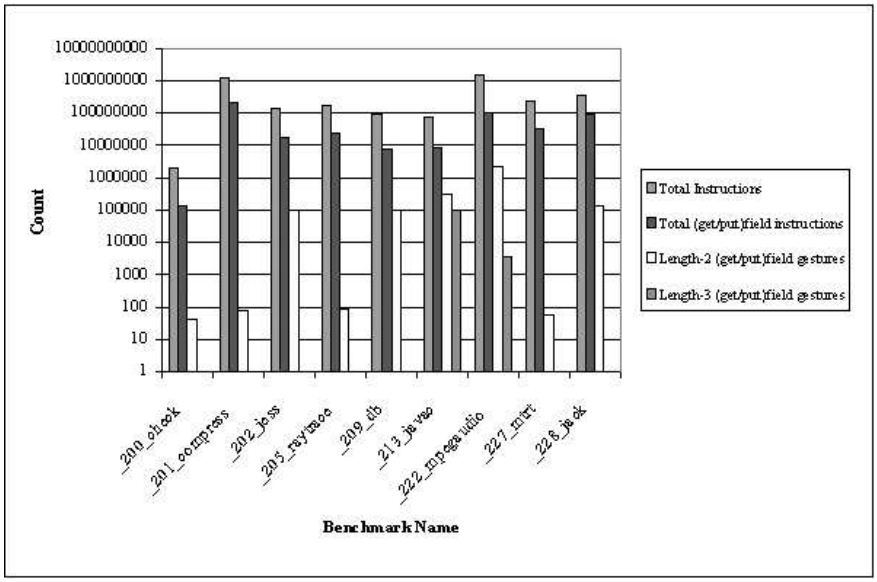


Figure 2: Number of Gestures Found in Size-10 SPECjvm98 Benchmarks with DYGS, by Length

executed in the gesture (*i.e.* a `getfield` or `putfield` Java bytecode instruction). The resulting complete gesture is then added to the distribution according to its current length.

Determining if an instruction can continue a gesture is a somewhat complex process [5], but it can be summarized as follows. A contiguous sequence of executed instructions could form a single gesture, but it is also possible that the sequence contains multiple gestures. This is because the components of a given gesture can be interrupted by other instructions, including instructions that comprise other gestures. DYGS keeps track of the runtime stack's contents so as to remember whether any intermediate computation is in fact a a gesture.

### 2.1 DYGS Results

As can be seen in Figure 2, 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, `javac` and `mpegaudio`, contained gestures of length three, and all contained gestures of length two.

The benchmarks with the longest gestures also tended to have the highest frequency of gestures. This high-frequency group contained the `jess`, `jack`, and `db` benchmarks, in addition to the two already mentioned. In the case of some of the larger benchmarks, this frequency occasionally reached past seven decimal places.

## 3 Dynamic Analysis of C programs with SimpleScalar

Thus far, our analysis has focused exclusively on Java programs. Java's referential transparency and its stack-based Java Virtual Machine (JVM) are particularly well-suited to gesture discovery and the optimization described in Section 5. However, it is worthwhile to investigate the potential of gestures for other languages as well.

We chose to investigate C programs, based on that language's widespread use, and the numerous differences C programs have with Java programs. To perform such analysis, 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* [10], 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

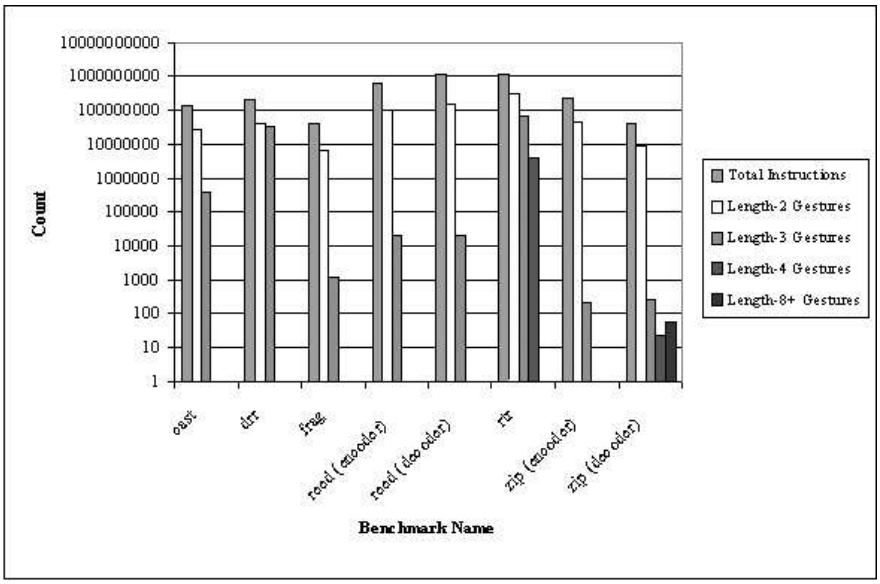


Figure 3: Number of Gestures Found in CommBench Benchmarks with DYGS-SS, by Length

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.

SimpleScalar is a register-based architecture: memory load- and store-instructions operate on a group of registers. When finding gestures in Java bytecode, we could examine the stack (in particular, its top) to determine whether the next instruction continues a given gesture. Extending this idea to registers, we keep track of the gesture state of each register, since any register could develop an address used for a gesture. We therefore transfer gesture-state from a source register to a destination register when a memory load instruction occurs. We refer to the process of keeping track of these values as *register shadowing*. Specifically, we keep track of the longest gesture that the value in each register could currently comprise. Upon encountering a memory-accessing instruction that uses a register as its source, the gesture length associated with that register is incremented and moved to the instruction's destination register. Conversely, when any other type of instruction accesses a register, a gesture in progress could not continue beyond that point, so the register's current gesture length is recorded and then reset to 0.

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

## 3.1 DYGS-SS Results

We examined gestures in the CommBench benchmark suite [12], and found that all programs 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 (see Figure 3).

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, this benchmark was found to have several long gesture types, up to a length of 128 instructions (see Figure 4). 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

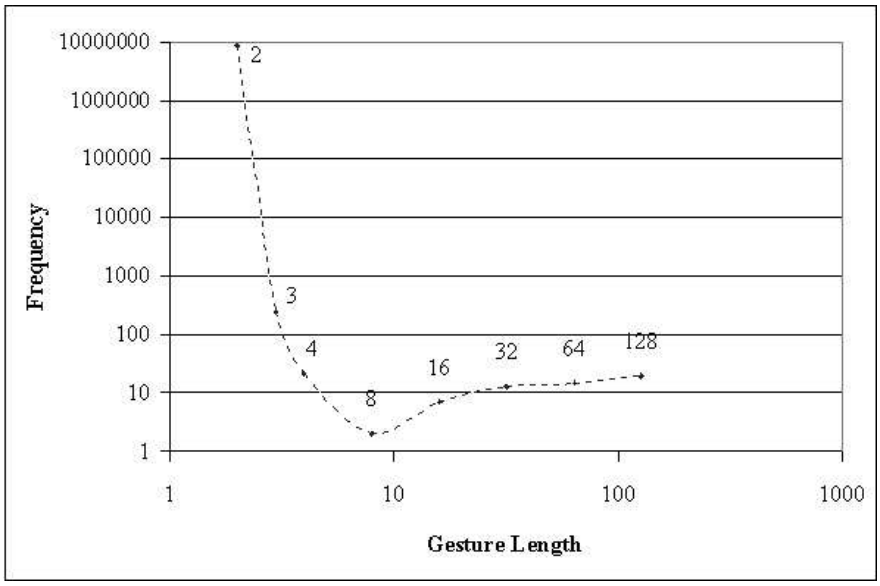


Figure 4: Gesture Distribution by Length in Zip Decoder Benchmark

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 that were found in this benchmark.

# 4 Static Analysis

We next turn to finding gestures at compile-time, using various forms of *static* analysis. While dynamic analysis has shown the presence of gestures, unless they can be found statically, they can not be transformed into gesture instructions. We performed static analysis in two ways:

1. The sequence of bytecode instructions for a given package was examined using a filter on the `javap` disassembler output. This approach can find only those gestures that appear without interruption.

2. Data flow analysis was performed on a method to find gestures that correspond to noncontiguous instructions in the bytecode file.

## 4.1 Static Analysis Based on Adjacent Instructions

Our first experiment finds gestures using the `javap` [8] utility that comes packaged with most Java installations. With the `-c` command-line option comes a complete listing of the opcodes for each method of a Java class file. In this case, a gesture is any group of consecutive `getfield` opcodes of length greater than one that is surrounded by non-`getfield` opcodes. We wrote a Perl script to process the output of `javap`, computing the number and length of gestures in a set of `.class` files.

We experimented with the SPECjvm98 benchmarks as well as Java's `io`, `lang`, `math`, `net`, `security`, `text`, and `util` packages. The results are shown in Figure 5.

None of the results contained any indirection chains longer than length two, but this behavior was expected. This approach did not detect sequences ending with a `putfield` instruction, because a chain ending with a `putfield` instruction has a `push` or `aload` instruction in the middle that puts the value to assign the field onto the stack. Because the script only finds consecutive `getfield` or `putfield` instructions, the extra `push` is interpreted as a break in the chain.

| Package | Double Chains |
|---|---|
| **check** | 0 |
| **compress** | 0 |
| **jess** | 0 |
| **raytrace** | 3 |
| **db** | 0 |
| **javac** | 216 |
| **mpegaudio** | 102 |
| **mtrt** | 0 |
| **jack** | 14 |
| **checkit** | 0 |
| **java.io** | 4 |
| **java.lang** | 0 |
| **java.math** | 0 |
| **java.net** | 0 |
| **java.security** | 0 |
| **java.text** | 44 |
| **java.util** | 3 |

Figure 5: Gesture Counts using javap -c

## 4.2 Static Analysis using Data Flow Analysis

To find longer gestures, we developed a data flow analysis tool called *Seekr* that finds gestures in a more robust manner. By using the Clazzer [7] library, we could take advantage of data flow analysis [1] and its modeling capabilities to find indirection chains that were split by other byte code instructions, specifically chains ending with a `putfield` [6].

For most compilers, `putfield` chains are generated as follows: `getfield getfield aload putfield`. This is a triple indirection chain, but our `javap`-based approach counted only the two `getfield`s as part of a chain. With Clazzer, we can construct an instruction graph that shows the "parameters" of each instruction. Thus, the above chain becomes `putfield(aload, getfield(getfield))`.

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

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

Using one possible execution path, where the `boolean` variable is true, the `int result` is set to `b.f.integer`, which is a double indirection. To recognize this as a valid gesture, we need to use data flow analysis.

Clazzer gives us the ability to break a program down into vertices in a graph which are connected based on possible execution paths. These vertices, called `StackMonitorVertices`, can contain information about the size and contents of the stack for each path.

StackMonitorVertices resolve stack operations between `getfield`s and keep track of what value a `getfield` is actually being called on rather than just what instruction comes before it in the byte code. They also display the instruction chains in a Lisp-type format. The double-indirection of `b.f.integer` then becomes `getfield integer (getfield f(new Bar))`. This greatly simplifies the work at each node to find gesture lengths. Figure 6 shows two possible `getfield` chains, one with just one data flow path and one with two.

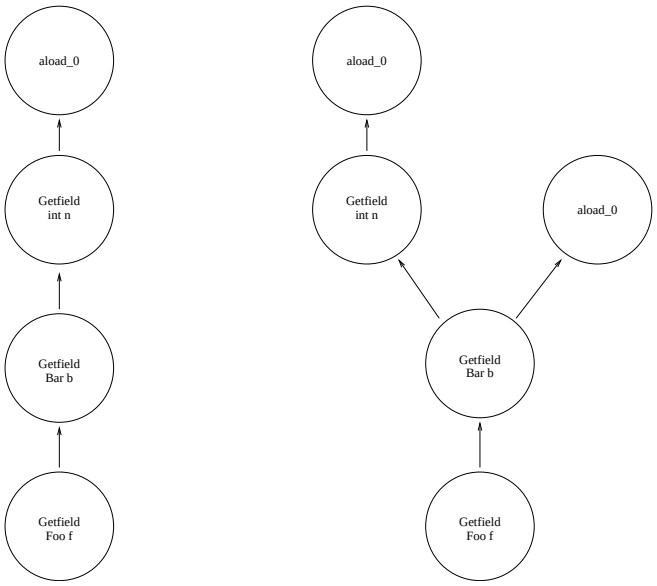


Figure 6: Two Getfield Chains: One With One Possible Data flow Path and One With Two

| Benchmark | Classes | Chains | |
|---|---|---|---|
| | | Double | Triple |
| **check** | 65 | 48 | 18 |
| **compress** | 49 | 3 | 13 |
| **jess** | 144 | 26 | 32 |
| **raytrace** | 76 | 7 | 16 |
| **db** | 58 | 3 | 13 |
| **javac** | 155 | 183 | 49 |
| **mtrt** | 76 | 7 | 16 |
| **jack** | 88 | 18 | 24 |
| **checkit** | 59 | 3 | 16 |
| **JBB** | 91 | 58 | 32 |

Figure 7: Benchmark Statistics using Data Flow Analysis

We ran the Seekr program to gather some general data about the Java benchmarks and to obtain a more accurate count of length-2 and length-3 gestures. These counts include `putfield` instructions in the chains. The results are shown in Figure 7. Notice the improvement over the results from `javap`. Not only do triple indirection chains exist, but in many of the benchmarks they are more plentiful even than double indirections.

# 5 Macro Optimization

Prior to executing a program, our approach calls for transmitting all necessary macros to the storage subsystem. The number of macros actually needed apparently depends on the success of our static approach in finding gestures. However, the actual composition of the gestures is dependent on how structures are organized in memory. For example, consider a structure with two components, `a` and `b`. If `a` occurs first, then a gesture involving `a` would specify offset 0 from the start of the structure. However, if the components were reversed, an offset of 4 might be required.

Thus, the gestures present in a program depend on the organization of structures. For some programming languages, notably Java, it is not possible to develop references based on structure organization. For those languages, we

| Class Foo | |
|---|---|
| Foo | a |
| Foo | b |
| int | c |
| double | d |

Figure 8: Example Field Alignment

| |
|---|
| Foo.a.c |
| Foo.a.d |
| Foo.b.a |
| Foo.b.d |
| Foo.a.a |

Figure 9: Example Instruction Table

have the opportunity to reorganize structures so as to minimizes the number of necessary macros. For example, if a given structure can be reorganized so that gestures involving that structure are already covered by macros previously used, then no extra macro is needed for that structure.

Unfortunately, the problem of minimizing the number of necessary macros is NP-hard [5]. We next present a heuristic and show how well it works on the benchmarks we studied.

## 5.1 Greedy Heuristic

Assume you have all the classes used by a program along with a list of their fields. For example, Figure 8 shows a sample structure `Foo`.

Assume you also have a list of the indirection instructions called by the program. Figure 9 shows a sample instruction table for `Foo` as defined in Figure 8. In this example, the initial `Foo` is actually an instance of a `Foo` object, not a static method call.

Now make a chart of all fields in all structures, and for each field count up how many times that field appears as the first field in a chain, the second field in a chain, and so on. For example, Figure 10 shows the properly-constructed chart for `Foo` defined in Figure 8 and the instructions stated in Figure 9.

Now use this chart to reorder the fields of each structure using the following rules:

- All fields that are not referenced in any chain go at the end of the structure, in any order.
- Of the fields that are found in a chain, all reference fields (i.e. those that have a non-primitive type) come before primitive fields.
- Order reference fields first based on the first column of the frequency chart and break ties using the second column. For programs with longer chains, ties in both columns could be broken using the third column.
- Order primitive fields in the same manner once all reference fields have been ordered.

So in the previous example, `Foo` would have the alignment shown in Figure 11. Notice that because c and d are both primitives and both had a first-column frequency of zero, the tiebreaker was their frequency as the second field in a chain.

Once fields are reordered, the structures need to be rewritten using the new ordering. There are a number of open-source tools available that allow the editing of Java class files, including Jclasslib [3].

We implemented our greedy heuristic in the Seekr program and tested its performance against

- an exhaustive search for an optimal field alignment, and
- a randomly-generated field alignment.

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

Figure 10: Example Frequency Chart

| Class Foo | |
|---|---|
| Foo | a |
| Foo | b |
| double | d |
| int | c |

Figure 11: Example Field Alignment after Algorithm Execution

For the exhaustive search for an optimal alignment, we were able to get upper and lower bounds on the number of intelligent macros for most of the Java benchmarks. However, for the larger benchmarks, the sheer number of possible combinations of field alignments was so large that it would have taken centuries to complete. For these, we ran the program for a couple of weeks and took the minimum and maximum number of macros as of that point. Note that this does not give us any guarantee on the real upper or lower bounds, but is simply an estimate given the time constraints.

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

We ran the three reordering algorithms on the SPECjvm98 benchmarks and counted the resulting number of macros for each. We also counted the number of macros to cover the compiled program right out of the box (OOB). Figure 12 shows the results of these tests.

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

Runtime is much shorter for greedy than for exhaustive on most of the benchmarks, and the heuristic comes very close to the measured minimum for all of them. Because the greedy strategy performs comparably to exhaustive and gives us a guaranteed result whereas random does not, it is an effective algorithm for reducing the number of macros needed to cover all gestures in a program.

# 6 Memory Macro Simulation

In this section we present simulation results that indicate the time that might be saved using our approach. The time savings should translate into power savings, since fewer trips between the CPU and storage subsystem should save power. We are currently investigating power savings, and report only on time in this section.

To study the time savings, we developed a simulator that lets the user specify various high-level characteristics of the "machine" that is being simulated, including information about the average execution times of several instruction classes, such as `getfield`s, `putfield`s, and instructions that do not access memory. Analysis of simulated program execution times under different sets of these characteristics can allow better understanding of which environment and program types are most conducive to incorporation of memory macros.

| Benchmark | Macros | | | | |
|---|---|---|---|---|---|
| | OOB | Maximum | Random | Greedy | Minimum |
| **check** | 19 | 24 | 24 | 17 | 17⋆ |
| **compress** | 9 | 9 | 9 | 8 | 8 |
| **jess** | 17 | 13 | 16 | 10 | 10 |
| **raytrace** | 13 | 14 | 14 | 11 | 11 |
| **db** | 8 | 9 | 10 | 8 | 8 |
| **javac** | 41 | 34 | 49 | 28 | 26⋆ |
| **mtrt** | 11 | 14 | 15 | 11 | 11 |
| **jack** | 14 | 12 | 15 | 11 | 11 |
| **checkit** | 9 | 12 | 12 | 9 | 9 |
| **JBB** | 30 | 29 | | 20 | 21⋆ |

Figure 12: Algorithm Comparison. The minimum runs marked with a star (⋆) did not complete, so the numbers reported for those entries were the smallest number of macros found in two weeks' elapsed time.

### 6.1 Simulation Example

Various datasets can be generated with the simulator 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 the simulator 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 our Program Generator to generate a 10,000-element gesture-length format (GLF) file with those approximate characteristics.

This GLF file can then be loaded into the Simulator. 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 the PIM takes an additional 2 ns over standard memory retrieval time.

The Simulator 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.

### 6.2 Simulation Results

An example set of macro processing times calculated with this method for each of SPECjvm98 benchmarks is listed in Figure 13. We used measured approximations for the execution times of each type of instruction on a Sparc machine running 4 300Mhz processors.

You can see from these numbers that using reasonably-accurate estimations for instruction processing times, we expect to see a speedup for all benchmarks and a significant speedup for the larger ones.

## 7 Related Work

The idea of reference gestures is akin to Proebsting's *superoperators* [9]. In a sense, our gestures are parameterized superoperators, where the gesture's parameter specifies the starting address of the gesture. The value of superoperators in compression has also been studied [4], but the extent to which they can reduce CPU-to-storage-system traffic had not been studied. Mowry *et al.* have studied the effect of compiler-based prefetching for recursive data structures. Our techniques are highly applicable by translating the prefetching into macro gestures communicated to the storage subsystem. Dolby and Chien discussed using object inlining do reduce allocation and pointer dereference costs [2], but our idea aims to improve runtime without increasing code size.

| Benchmark | | Simulated Execution Time (sec) | | |
|---|---|---|---|---|
| **Name** | **Size** | **without Macros** | **with Macros** | **Savings (%)** |
| jess | 1 | 0.1787 | 0.1646 | 7.89 |
| jess | 10 | 2.3605 | 2.3237 | 1.56 |
| jess | 100 | 21.8655 | 20.9771 | 4.06 |
| db | 1 | 0.0471 | 0.0446 | 5.31 |
| db | 10 | 1.6514 | 1.6268 | 1.60 |
| db | 100 | 140.0346 | 137.4501 | 1.85 |
| javac | 1 | 0.1906 | 0.1802 | 7.34 |
| javac | 10 | 1.6690 | 1.4767 | 11.52 |
| javac | 100 | 33.2832 | 28.9092 | 13.14 |
| mpegaudio | 1 | 1.8902 | 1.8456 | 2.36 |
| mpegaudio | 10 | 18.4694 | 18.1617 | 1.67 |
| mpegaudio | 100 | 182.4440 | 179.1127 | 1.83 |
| jack | 1 | 3.4778 | 2.3487 | 32.47 |
| jack | 10 | 3.4848 | 2.3468 | 32.66 |
| jack | 100 | 64.2033 | 45.2139 | 29.58 |

Figure 13: Simulated Execution Times of SPECjvm98 Benchmarks with and without Memory Macros

# 8 Conclusion

We have shown that storage-referencing gestures occur in Java and C programs, and that such gestures are easily discovered at compile-time. Even though most are relatively short (depth-2), translating them into macro instructions could reduce the traffic across the CPU–memory bus by a factor of two. The power and cycle savings could be significant for embedded and low power systems. We have considered optimizing the number of distinct macros needed to represent a program's gestures, and we have presented a simple but effective heuristic that can reduce the number of distinct gestures.

# References


[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[2] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 345–357, Vancouver, British Columbia, Canada, 2000.

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

[4] William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 148–155. ACM Press, 2001.

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

[6] Christopher R. Hill. Static analysis of memory-accessing gestures in java. Master's thesis, Washington University in St. Louis, 2003.

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

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

[9] Todd A. Proebsting. Optimizing an ansi c interpreter with superoperators. *POPL*, 1995.

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

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

[12] 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.