Refactoring gcc Using Structure Field Access Traces and Concept Analysis

Robert W. Bowdidge
Apple Computer
1 Infinite Loop
Cupertino, CA 95014
bowdidge@apple.com

ABSTRACT

Refactoring usually involves statically analyzing source code to understand which transformations safely preserve execution behavior of the program. However, static analysis may not scale well for large programs when analysis results are too general, when tools for analyzing the source code are unwieldy, or when the tools simply do not exist. In such cases, it can be simpler to analyze the program at runtime to gather answers needed for safe code changes. I show how dynamic data can guide refactoring of a single data structure into a hierarchy of classes. Specifically, I show how I refactored the gcc compiler to cut its use of heap memory. In order to partition the declaration data structure into more efficiently-sized parts, I used data structure field access traces to automatically identify how the data structure might be refactored. I also identified other potential refactorings of the data structure using concept analysis. These results then guided by-hand modifications to the compiler. I finally evaluated what size test cases would be needed to gather adequate information to correctly perform the refactoring. The case study showed the refactoring could be performed with the dynamic information, but without traces from an exhaustive set of test cases, some fields would be moved incorrectly.

Categories & Subject Descriptors

D.2.4 [Design Tools and Techniques]: Modules and interfaces; D.2.5 [Testing and Debugging]: Tracing; D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering

General Terms

Design, Measurement.

Keywords

case study, gcc, meaning-preserving restructuring.

1. INTRODUCTION

Restructuring and refactoring tools[7][9] are maintenance tools intended to allow a programmer to transform a program from one design to another while maintaining properties from syntactic correctness of the code to output behavior. Such tools usually rely on static analysis—parsing, type analysis, and dataflow analysis—to decide which transformations can be applied safely. Useful static analysis tools are not always available. Tools may not handle large source code bases, coding styles could cause results to be too broad and imprecise, tools may not exist for the particular language, or merely specifying configuration details needed for parsing the source code may be non-trivial.

In my case, I needed to shrink the memory footprint of gcc, an open source C compiler. My team had noted that the data structure for representing declarations in the code being compiled was wasteful. For some kinds of declarations, half the memory in the data structure was unused. I sought to break the declaration data structure into two, one structure for the "small" declarations that used few fields of the data structure, and one for the "large" data structures that used most fields. The key problem was understanding which fields were used in which contexts, both to maintain correct behavior of the compiler, and to identify the minimum fields needed in small declarations. Because this compiler was a large C program with a particularly complex configuration, I had few tools to analyze the source code in appropriate detail.

My solution was to rely on runtime data to understand how the declaration data structure's fields were accessed. I used the data in two ways. First, I used the data to programmatically decide which fields could be moved safely based on the preconditions of the refactoring transformation. Second, I applied concept analysis [6][12] on the field access data to suggest a hierarchy of data structures that would save additional space.

This real-world example shows how logging field accesses at runtime can provide the data to answer if a refactoring transformation is safe. I will first describe the gcc compiler and why shrinking memory use was important. Next, I will explain how I gathered field access data in gcc. The analysis results are used to directly test preconditions of the refactoring transformation, and automatically suggest a likely design. I also examine the results manually to identify fields included in the small declaration because of unneeded accesses in the source code. I will describe how concept analysis discovered potential class hierarchies for further space reductions. Finally, I will evaluate whether the dynamic information was sufficient to make safe changes to the source code and whether the instrumentation approach provided sufficient coverage of all possible behavior.

2. THE PROBLEM: SPEEDING COMPILE TIMES

Apple uses gcc—the GNU C compiler—as the system compiler for Mac OS X. (Apple has shipped several versions of gcc; all work in this study was done on version 3.3 of gcc.) gcc's popularity comes from its open source development model and its adaptability to new processors and operating systems. Compila-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Dynamic Analysis (WODA 2005), 17 May 2005, St. Louis, MO, USA. Copyright 2005 ISBN # 1-59593-126-0 17/05/05 ... $5.00

tion times have been less important for gcc developers; most users of gcc find that it works well on their source code, and most users of gcc are used to UNIX compilers that are on par with gcc's performance. Developers who worked on Mac OS 9 with all-in-one IDEs (Code Warrior and Think C, for example) could be disappointed by gcc's compile times; the all-in-one compilers used a single process, hand-tuned code, and data structures shared between compilations to provide particularly fast compiles. Apple developers are often working on GUI applications, and rely more heavily on fast edit-compile-debug cycles to examine the effect of small changes.

One key cause of slow compiles is the amount of header files brought in when compiling a single source file. Because of the richness of the OS's application programming interfaces, Apple's system headers are particularly large; a simple application bringing in the top-level header for the Carbon library (only one of several large libraries in Mac OS X) eventually pulls in 100,000 lines of headers; these must either be compiled for each compilation unit, or must be precompiled so the processed data structures can be shared between compilation units. (Although such programs have been unusual in the UNIX world, more programs compiled with gcc—the Qt cross-platform GUI toolkit, web browsers, and the GNOME GUI toolkit—are similarly noting slow compile times and encouraging additional performance improvements.)

The size of header files directly affects gcc's memory use, and memory use eventually affects compile times. The program spends time managing the heap, waiting for memory to be paged in and out, and waiting for the CPU when all the needed data doesn't fit in the processor's cache. Because header files tend to be filled with declarations—functions to be found in libraries, constant values, and useful data structures—much of gcc's heap memory is filled with declarations. Declaration parse tree nodes exist for every global declaration in the source file and headers, and each exists until the end of compilation. gcc uses 21 MB of heap memory to compile only the Carbon library header files; about 1/3 of gcc's total memory use is for declaration parse tree nodes. Each declaration requires 116 bytes of memory, and has 64 fields.

Declarations can be one of several variants: constant declarations, variable declarations, parameter declarations, function declarations, type declarations, field declarations, and result declarations. (There are also template declarations, label declarations, and namespace declarations. They occur infrequently, and were ignored in this analysis.) Many of the fields in the declaration data structure are used only by specific declaration kinds; for example, the "abstract_origin" field is needed for parameter, variable, type, and function declarations, but not for constant, result, and field declarations. A significant number of all the declaration parse nodes created when compiling Carbon headers are for enum value declarations—constant declarations—because of a common code pattern for defining sequences of error values. We predicted that shrinking the size of constant declarations would cut memory use and speed the compiler.

3. SLIMMING CONSTANT DECLARATIONS

The solution that best fit gcc's coding style is to create two kinds of declarations—a smaller declaration containing only the fields needed by constant declarations, and a larger declaration structure containing all fields. gcc already concatenates structures to imitate the object-oriented idea of superclasses and subclasses; the first portion of the large structure (subclass) exactly matches the layout of the small data structure (superclass). This permits the same accessors to manipulate either data structure. (See Figure 1.) In terms of refactoring [5], I am creating a new subclass for an existing class, then pushing fields down from the superclass (the small declaration) into the subclass (the large declaration.) Moving the data structure's fields will not affect the running behavior of the code if done correctly.

To perform this refactoring correctly, one critical precondition must be met: a member being moved to the subclass must never be accessed by an instance of the superclass. (In gcc's terminology, a field may not be placed in the large data structure if it is ever needed by a constant declaration in a small declaration.) Two obvious approaches to testing this condition are either to refer to documentation or analyze the source code. The gcc header file defining the declaration structure describes the contents of each structure field for each kind of declaration. A given field might be used for one purpose in function declarations, for a slightly different purpose in variable declarations, and might be unused for constant declarations. Experienced gcc'ers expected this information would be reasonably accurate, but the comments usually said more about the cases where fields were used rather than where they were not. Nor was there any guarantee that some generic code would initialize a value otherwise unused by a particular kind of declaration.

Getting the information from the source code was less promising. The sheer size of the gcc sources (around 700,000 lines of source) and frequent manipulations of declarations made inspection or string-based tools unwieldy. gcc's stylized code and complex configuration could not easily be passed through the static

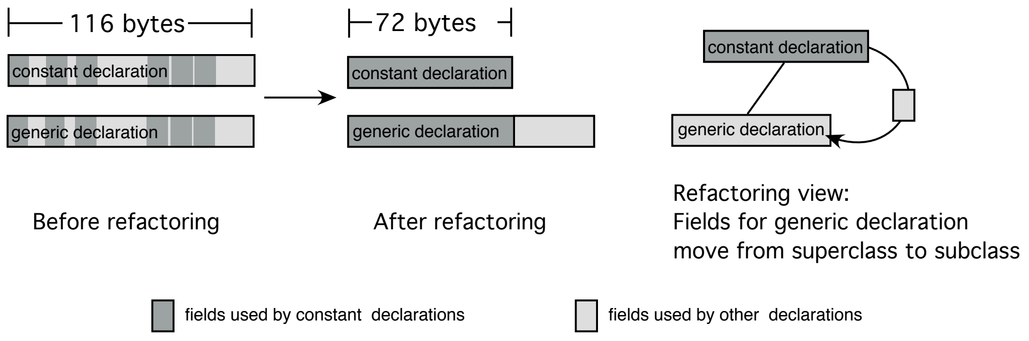

Figure 1: The left drawing shows the current data structures, where constant declarations and other declarations are the same size. The center drawing shows the state we'd like, with constant declarations being small, and other declarations being a superset. The right drawing shows the intent in terms of object-oriented design.

analysis toolkits I had on hand. gcc's heavy use of macros as replacements for inlined functions, parameterized hooks, and data structure accessors also make it non-trivial to parse. Deeper analysis would also be limited. All parse tree nodes are referred to with the "tree" type, so dataflow analysis would be required to identify even the places where declarations could be manipulated. Type analysis was also complicated because switch statements were often used to implement polymorphism, with different branches performing actions on the different kinds of declarations.

I instead chose to rely on runtime data over a set of representative programs to estimate which fields were used by which declarations. Runtime logging could not guarantee a particular field would never be accessed by a particular declaration, but in practice logging could provide more accurate information than any other technique available.

3.1 Instrumenting gcc

To test which fields were ever used by constants, I instrumented every access to a declaration data structure. (Because this logging was only for a specific refactoring task, the logging was kept as simple and easy-to-apply as possible.) Each access would cause a log entry naming the field and kind of declaration seen. I also logged the location of the access in the source code for a separate analysis. gcc's stylized macros made it easy to catch every access. Macros already tried to hide field accesses behind (effectively) inlined function bodies; only about 300 lines of code needed to be changed. As one might expect in a realistic, long-lived system, I also found a few manipulations of the fields were done directly, rather than through the macros.

To gather the field access information, I collected a training set of 183 source files from four realistic programs. Two are examples from Apple's developer tools that both use the Carbon library: SimpleText, a C program, and AppearanceSample, a C++ program. I also compiled rogue, a UNIX terminal-based game, and MiniRay, a ray-tracing renderer written in C++ using the Carbon library. Each program was between 5 and 10 KLOC (thousand lines of code). I compiled each with the instrumented compiler. Logging slowed compiles down by a factor of three. Generating all the logs took around two minutes on a two processor PowerMac G5. Log files for realistic source files recorded about 4 million events and required 200MB; summarizing this data by declaration kind, field name, and location in the source resulted in 40KB files for each source file compiled. Each of these compressed logs was saved, for a total space for all logs of 16 MB.

With the logging information, identifying the fields required by constant declarations was trivial. The precondition for moving fields out of the small (constant) declaration data structure and into the large declaration data structure is that no instance of a constant declaration must access that field. I wrote a script to find the inverse of this: "which fields are accessed for a declaration of type x?" The result of this question for constant declaration is the list of fields that must be in the constant declaration data structure. The results were generated with no human intervention; if I had a refactoring tool that could modify C source code, the fields could have been moved automatically. Without such a tool, I took the results for constant declarations, and applied the changes by hand based on the list.

The simple list did not capture one aspect of the C programming language: data structure packing rules. The declaration data structure has about 25 fields that are single-bit and multi-bit fields. On Mac OS X, data structures are arranged on four byte boundaries. If the flags were divided across two data structures, an additional four bytes might be wasted. I moved all the flags into the "small" data structure to avoid packing problems.

By moving fields into the "small" declaration according to the list of fields generated by the logging, gcc now required only 76 bytes for each constant declaration rather than 116 bytes for all other declarations.

3.2. Removing Accidental Fields

Like refactorings based on static analysis, the field access information also suggests refactorings based on the actual behavior of the code, rather than the programmer's idealized view of its structure. In gcc's case, the list of fields needed by constant declarations included some surprises. The size field, size_unit field, mode field, and rtl (pointer to the intermediate representation for code generation) were supposedly accessed, even though they made little sense for constant declarations. A few accesses performed indiscriminately for a "generic" declaration would keep a fully automatic system from removing these "accidentally used" fields.

To help the engineer performing the refactoring, a refactoring tool that can check "will field x ever be used by class y?" also needs to be able to explain the reasons for that answer: "where do uses of field x for class y occur"? As part of logging, I gathered the function, source file, and line number for each access to a declaration data structure; with this, a simple script lists places in the source code where accesses to a particular field by a particular kind of declaration occur.

By examining the use sites, I determined that many of the fields could be moved out of constant declarations if the source code of the compiler was changed. The rtl field was only set during initialization of the constant declaration. Size fields were set late in the compile, but their values were never read. After eliminating the unneeded operations and rerunning the analysis, a new list of fields for constant declarations was recommended. By implementing these changes in a new version of the compiler, constant declarations now only took 60 bytes.

Some of the diagnosis of these "accidental" field uses could be done automatically. Fields accessed only once per constant declaration created were likely initialized but never read. Fields only accessed on a few objects might indicate a bug or an uncommon operation. Similarly, the field analysis can hint at the algorithms being used. Fields accessed twice as many times as there were objects indicated initialization and a single access; larger access counts indicated tight loops over all objects or a need for frequent access. Such information could hint to a programmer which fields may deserve examination for performance improvements.

One particular performance improvement was found with later analysis of the field access data. When compiling just the Carbon headers in C++, there were many more function declarations created than when compiling the same headers with C, and the names of functions were accessed significantly more times. This increase was caused by the fact that structures and classes are the same concept in C++. When each structure/class is defined, the C++ standard says that six functions must be created—the plain constructor, copy constructor, and destructor, with separate versions of each for manipulating an object of the same type or an object derived from the current type. Because most of the structure definitions are never used in the source, the implicit functions are not used either. In gcc version 4.0, these functions are now created lazily when they are first needed in code—one of many changes that sped gcc up by 35% in 2004.

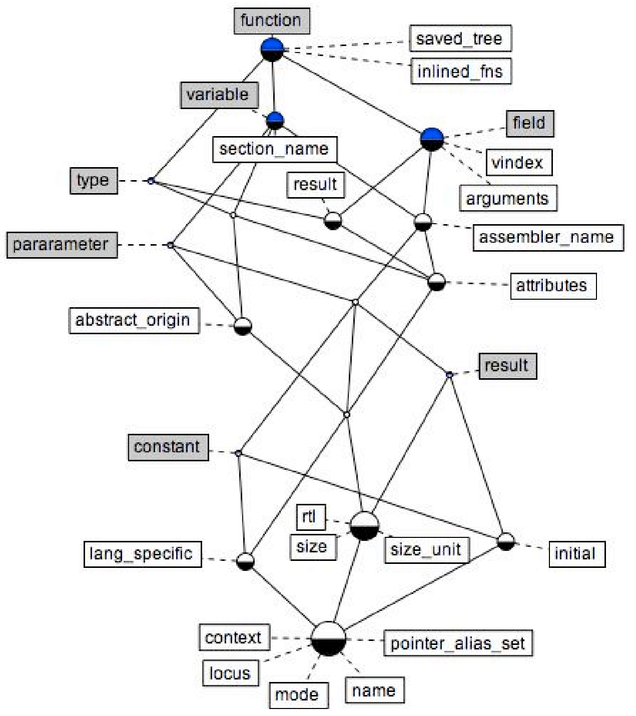

Figure 2: Concept lattice showing potential hierarchies of kinds of declarations. Gray boxes represent declarations, white boxes represent fields. Fields between a declaration kind and the bottom of the graph are used by that declaration. Fields representing boolean values have been removed for clarity.

3.3. Identifying A Hierarchy Of Declarations

By placing constant declarations in their own, smaller, data structure, gcc's memory use was reduced. My obvious next question was "can any other kinds of declarations be broken off into smaller data structures?" Repeating the "which fields are accessed" query for other declarations showed that several only used part of the entire declaration data structure. An obvious candidate was the parameter declaration, also typically responsible for 30% of the declarations allocated in typical compiles. However, what kinds of declarations might use a similar, and small, set of fields?

Concept analysis gave me a way to identify potential hierarchies. Concept analysis is a mathematical technique to detect similarities and hierarchies in a set of objects[6]. At its simplest, concept analysis converts a matrix of objects and attributes to a graph (more precisely a lattice—a connected graph with a recognized top and bottom node) that clusters items with similar attributes in the graph.

Snelting and Tip[12] recognized that concept analysis could be used for analyzing a program's class hierarchies. By taking a set of Java classes and noting which instance variables or methods in each class were used, they could use concept analysis to generate a similar but synthesized class hierarchy that could be distributed in a smaller file.

In my case, I used concept analysis for the analogous problem of building up a hierarchy by noting which kinds of declarations actually used similar fields of the declaration structure. Figure 2 shows the concept lattice generated from the field access data. (The image was generated by Concept Explorer[17].) In this lattice, nodes are associated with kinds of declarations (the objects, drawn with shaded blocks). Attributes are associated with fields of the data structure (drawn with white blocks). For any data structure field, the declarations which access that field are along paths leading to the top of the lattice. For example, the arguments field is only used by field and function declarations because the only path to the top goes through those two nodes. The result field is used by type, field, and function declarations. The initial field is used by every kind of declaration except type declarations.

The concept lattice not only identifies relationships between data structure fields and kinds of declarations, but also shows a rough hierarchy. Because the result declaration is along at least one path from the parameter declaration to the bottom of the graph, the two declarations use some of the same fields. This suggests a subclass relationship—many of the fields in the superclass (result declaration) are also needed by the subclass (parameter declaration). The relationship is not exact; some fields needed by each are not needed by the other. However, the groupings and relative positions of objects can hint at potential class hierarchies.

As in the constant declaration example, gcc's design and use of the C language encourages extending data structures by concatenating structures, rather than by creating a tree of subclasses. Thus, the most straightforward approach creates a new "medium" declaration that extends the small constant declaration, but borrows fields from the large declaration. The concept lattice suggests several potential groupings. Two choices are:

- small data structure: constant declarations
- medium: parameter and result declarations
- large: variable, type, field, and function declarations
  or

- small: constant, parameter, and result declarations
- medium: type declarations
- large: variable, field, and function declarations

This kind of refactoring choice is generally a design decision for a human. With distribution of kinds of declarations for typical programs, I could predict which choice gives the best decrease in memory. My choice was to leave constant declarations in the small data structure, place the parameter and result declarations in the medium data structure, and the rest in the large data structure. This resulted in a compiler with a 60 byte small declaration, 96 byte medium declaration, and 116 byte large declaration.

4. RESULTS

There are two ways to test whether refactoring using runtime data succeeded in this particular case. First, because this is primarily a case study, I will describe how the resulting compiler worked, and whether any cases were missed. Second, I can identify how many fields could potentially be detected, and thus compare the results from the small set of test cases to optimal results.

4.1 Did the Refactoring Succeed?

This work, as a case study, focuses on a single experience: refactoring gcc. To test whether the compiler still worked after refactoring the compiler, I compiled two large (500KLOC) C++ applications, Finder (the Mac OS X file browser) and the Qt cross platform GUI library[15]. These tests turned up two problems. First, one field ("attributes") were used by parameter declarations, but had not been been seen in the training set. The omission of attributes occurred because the training set did not use C++ templates heavily; comments in the gcc sources did indicate that parameters would use the "attributes" field. There were also two cases where I moved fields incorrectly—a reminder of how errorprone refactoring by hand can be, and why tool support is so desirable.

By applying the changes suggested by the field use analysis and fixing the found problems, the resulting compilers bootstrapped (compiled themselves successfully) and compiled all the programs in our compile timing test suite.

Based on the gcc refactoring experience alone, the idea of refactoring based on runtime field analysis appears interesting, but not error-free.

4.2 Accuracy of Suggested Refactoring

The second way to test if dynamic information can guide refactoring for gcc is to identify how much effort is required to discover all possible field accesses, and how well the training set discovered the fields accessed by constant, result, and parameter declarations. Success can be measured by identifying optimal results, and comparing results from different test cases to these results. I can approximate optimal results by analyzing the compiler as it compiles a much larger set of programs—either gcc's own test suite, or a large set of realistic programs.

First, I ran the field logging compiler on gcc's own test suite, assuming that test suite would fully exercise the compiler. gcc's test suite runs about 15,000 separate compiles, taking about 90 minutes with an unmodified compiler. The test suite showed 23 fields were accessed by constant declarations, and 27 fields were accessed by parameter declarations.

As noted in section 3, I originally planned my refactoring of gcc by running the instrumented compiler over the "training set" —several non-trivial programs measuring about 20 KLOC (thousand lines of code). These compiles take about 30 seconds with an unmodified compiler. The training set runs discovered 21 fields accessed by constant declarations, and 26 by parameter declarations. Compared to the gcc results, I missed two field accesses to constant declarations ("arguments" and "abstract origin") and two field accesses to parameter declarations (”attributes” and "saved_tree" .)

Finally, I wondered how well the gcc test suite and training set exposed field accesses as compared to real programs. To test this, I also ran the field-logging compiler on 21 projects from the Mac OS X operating system, representing six categories of language and program purpose. Compiling all these projects takes about 90 minutes with an uninstrumented compiler. For each project, I collected the number of unique fields accessed by constant declarations and parameter declarations. Table 2 summarizes the results for each project, for each category of project, and for all projects.

The real programs discovered two fields used by parameter declarations not found by the training set ("attributes" and “saved_tree”), but missed the same two fields used by constant declarations that were found by the gcc test suite but missed by the training set ("arguments" and "abstract_origin".) Breaking down the results by programming language and type of program, no category of program found all 21 references to constant declarations and 27 references to parameter declarations discovered overall by the real programs—each category of projects found a different set of fields. The differences between categories of projects is not surprising, as each compiler—C, C++, and Objective C—shares only some code with the other compilers, and vary in how they manipulate data structures.

Why did the regular programs and training set not find all possible field accesses compared to the gcc test suite—were the programs just not representative, or were the missing fields special in some way? One possibility is that the fields are very rarely accessed. I searched for fields by a particular declaration which were accessed in less than 1% of the gcc test cases using that declaration. Three fields matched this criteria for constant and parameter declarations: "ignored_flag", "abstract_origin", and "saved_tree"; two of those three fields were missed by the training set. The third field missing from the training set, "arguments", occurred in a bit more than 1% of the gcc test cases.

On closer examination, these rare accesses appear to either be accidental cases or bugs. "abstract_origin" is accessed by con-

Table 1: Table 1: Fields of constant and parameter declaration data structures accessed during the compilation of 21 projects from Mac OS X. The bottom entries of the table show the number of fields found when analyzing compilation of the training set of programs used for the gcc refactoring, and the number of fields discovered during runs of the gcc test suite.

Program Name	LOC	number of compiles	fields used by constant declaration	fields used by parameter declaration
C Libraries			15	23
CLib 1	255,000	97	15	22
CLib 2	32,000	43	0	0
CLib 3	2,000	34	15	22
C Unix-style Programs			17	23
CApp 1	6,000	11	14	21
CApp 2	1,000	7	15	22
CApp 3	8,000	33	14	22
CApp 4	2,500	11	15	22
CApp 5 (perl)	157,000	428	15	23
CApp 6	28,000	81	17	23
CApp 7 (Python)	272,000	499	17	22
C++ device drivers			20	25
Driver 1	15,000	19	19	24
Driver 2	1,000	10	19	24
Driver 3	11,000	12	18	24
Driver 4	236,000	66	20	25
Driver 5	32,000	75	19	25
C++ Libraries			21	25
C++ Lib 1	226,000	865	21	25
C++ Lib 2	77,000	243	20	24
C++Lib 3	49,000	55	20	24
C++ Applications			21	25
C++ App 1	247,000	380	21	24
C++ App 2	53,000	67	20	24
Objective C Libraries			17	24
ObjC Library 1	158,000	548	17	22
ObjC Library 2	489,000	105	17	24
ObjC Library 3	25,000	37	15	23
All programs			**21**	**27**
Training programs		**119**	**21**	**25**
gcc Test Suite		**15,000**	**23**	**27**

stant declarations in two of 15,000 test cases. In both cases, the accesses occur in code intended for duplicating inlined functions, suggesting a potential bug. The uses of "ignored_flag" by constant declarations and "saved_tree" by parameter declarations both occur in rare code paths in debug symbol generation, and occur in generic code checking the value in fields indiscriminately.

These results suggest that for refactoring data structures in gcc, relatively small test cases—either the training set or a selection of real projects—can discover most but not all field accesses. Identifying the rarely-accessed fields requires exhaustive tests with a much larger test suite. The rarely-accessed fields may be particularly interesting, hinting at bugs or poor code behavior. In any case, the rarely accessed fields must be identified to perform a correct source code change; although the quick tests may be suitable for planning a refactoring operation or prototyping and checking performance of a change, the exhaustive tests must be performed at least occasionally to verify the transformation made to the source code is safe.

4.3 Measuring gcc's Speedup

As for the results of the case study of refactoring gcc, the refactoring work did not speed up the compiler. For projects that read significant amounts of headers (or for compilation of precompiled headers), memory use for parse tree nodes dropped around 3-5% with the constant declarations refactoring, and 6-8% with the medium declaration refactoring. These numbers match expectations—if 33% of all declarations are constants, and 33% of parse tree nodes are declarations, then any shrinking of constant declarations would decrease that 11% of memory holding constants. Compile times either did not improve, or improved at best about 2%. For projects that did not bring in significant amounts of headers, or used a precompiled header to read the header files only once, memory use and compile speedups were rare—only the precompiled header creation gained any memory advantage.

This result strongly suggests arbitrarily cutting memory use will not affect compile speeds linearly. One hypothesis is that gcc's problem is in the data structures that are touched, rather than the ones that are created. Previous performance efforts have removed loops that would iterate over all declarations; these have significantly sped the compiler. Reducing the total number of declarations produced (as seen with the lazily-constructed constructors) is also a better direction.

5. RELATED WORK

Static analysis is usually preferable for refactoring because it gives provable facts about what the code may ever do. Griswold[7] describes a Scheme refactoring tool that uses dataflow analysis to test that refactorings will not change the output behavior of the program. Snelting and Tip[12] use concept analysis and static analysis of Java bytecode to automatically identify and apply refactorings on Java class hierarchies. Their goal is to remove unused methods and instance variables, and reorganize class hierarchies to cut the size of jar files.

Refactoring is not restricted to using static analysis. Because Smalltalk programs could create references to names programmatically, the Refactoring Browser [10] tests rename transformations by running test programs and watching for references to the old name. Roberts[11] also describes a framework for dynamic transformations for the Refactoring Browser and suggests how a move instance variable transformation could be checked in Smalltalk, but provides few details on its use in practice or experiences with the transformations. The Daikon tool [4] analyzes a program's behavior to identify potential invariants across all variables; these have been useful for suggesting refactorings [8], and could also be used to guide refactoring operations as was done here for gcc. Daikon gathers information on a file-by-file basis (rather than focusing on a specific type of interest as here.)

Chilimbi, Davidson, and Larus [3] describe reordering or splitting data structure fields so frequently accessed data structures are colocated. These transformations are similar to what was done to gcc. Like Snelting, their work is more aimed towards automatic transformations of code rather than programmer-driven refactoring. Their SQL example is closer to this work (also showing speed improvements of 2-3%), but few details are provided about the issues they encountered performing the change.

Several other examples exist of using concept analysis to identify potential objects or class hierarchies[13]. Tonella and Ceccato[14] use dynamic analysis and formal concept analysis to find potential aspects in non-aspect-oriented code. They analyze function calling patterns to partition functions for a class into independent aspects. van Duersen and Kuipers[16] lexically analyzed a large COBOL program to find field accesses, then used that data to compare concept analysis and cluster analysis. Their work, like other legacy refactoring work [1][2], seek to identify related fields that should be clustered an abstract data type. This work differs by its focus on using refactoring transformations as a framework for the changes.

6. OTHER TRANSFORMATIONS USING DYNAMIC ANALYSIS

This paper shows how the "move field" transformation can be tested for correctness with information gathered at runtime. Can this same method be applied to other refactoring transformations?

There are obvious cases where dynamic information is not necessary for deciding if a refactoring transformation is safe. Renaming operations, changing the type of an variable or field, and wrapping field accesses in accessor methods can often be done with only static analysis quite well. Many such transformations can be performed in ways that cause the code to fail to compile if performed incorrectly[5], giving some assistance to a programmer trying to modify a large, legacy application.

Dynamic information would be better for transformations involving information that is hard to determine statically, especially those involving polymorphism, inheritance, and other dynamic decisions about what code will be executed. For example, moving a method (or virtual function in C++) from a superclass into only one subclass of many could change execution behavior if any of the other subclasses also use that method. Instead, a virtual function further up the class hierarchy could be called. Tracing all calls to methods of that name, then comparing traces before and after the modification, would indicate if different functions are called. Also in C++, several functions can have the same name but different numbers or types of parameters. Changing the type of a parameter could cause a different function to be executed. Again, tracing function calls before and after the modification could indicate changes in behavior, allowing the programmer to decide if the new behavior is reasonable and appropriate.

Specializing the type of a variable from a superclass to a subclass also could benefit from dynamic information; if the variable of type superclass only ever contains objects of type subclass, then changing the the type of the variable to subclass should be safe.

All these transformations would still need information about the class hierarchy to understand how to automatically instrument the code, and to understand whether the dynamic check is even required. For example, changing the number of parameters in a function requires no dynamic check if no other function with that name exists.

A key refactoring transformation is extract function-—taking a common expression occurring throughout the code, and replacing it with a call to a new function containing the expression as its body. Extracting a function can be unsafe because the subexpressions being parameterized will be executed before the rest of the function body. Detecting such side effects in programs with pointers is expensive for non-conservative results. For this transformation, there is no single variable or type to be instrumented (for a side effect could affect any variable in any function in the calling hierarchy of any code in extract function.) Tools such as Daikon [3], which identify invariants in the program by observing all variables at runtime, could identify such side effects. If an extract function transformation had side effects, the invariants for the program would change.

7. CONCLUSION

Refactoring large, long-lived systems written in common systems programming languages often must be done without the powerful refactoring and analysis tools available for more modern languages. Without such tools, it can be difficult for a programmer to decide which changes might affect the behavior of their program. Through a case study of refactoring the gcc compiler, this paper describes how results from instrumenting accesses to data structure fields can determine if a particular refactoring transformation—moving a field between structures—is safe. Specifically, I showed how the dynamic data could answer whether preconditions for safely moving a field between two data structures were met. I also noted that for gcc, some otherwise practical refactorings could be prohibited by unnecessary field accesses. Knowing where the accesses occur was necessary to help a programmer identify source code changes that could allow the refactoring transformation to be performed. I also demonstrated how concept analysis identified hierarchies of data structures in gcc and potential groupings of fields in multiple declaration data structures. Finally, I showed that analyzing gcc's behavior on small test cases found most, but not all, field accesses; the few rarely-accessed fields would often be detected only with more exhaustive test cases.

The work differs from previous work primarily by demonstrating how refactoring transformations using dynamic information might work with large, legacy programs, and highlighting some of the potential issues involved with refactoring such programs.

There are several directions for future work:

- Given the queries necessary in a static analysis-based refactoring tool, which queries can also be done with dynamic analysis. How accurate would they be, and how long would they take to run?
- Fowler[5] uses a cookbook approach to suggest how each transformation can be done manually. The refactoring demonstrated here could also be set up as a cookbook approach—"gather specific data, and these scripts will give you the answer you need." Can other refactoring transformations based on dynamic analysis be turned into cookbook approaches requiring little understanding of the analysis from the programmer?
- Can other kinds of refactoring transformations be performed efficiently on large programs? Which operations are infeasible using dynamic information?
- What tool assistance can be provided for instrumentation, data collecting, and refactoring for similar large programs?
- Which queries can be performed beforehand so that a user's request for a refactoring could be done with no new test runs needed? How would such requests be invalidated when code changes?

8. ACKNOWLEDGEMENTS

Thanks to Mark Chu-Carroll, Bill Griswold, Christine Halverson, and Ted Goldstein for providing helpful comments on an earlier draft.

9. REFERENCES

[1] Canfora, G.,Cimitile, A.,Munro, M., & Tortorella, M. (1993). Experiments in Identifying Reusable Abstract Data Types in Program Code. In IEEE Conference on Program Comprehension, (pp. 36-45). Capri, Italy: IEEE Computer Society Press.

[2] Casais, E. (1993). The Automatic Reorganization of Object Oriented Hierarchies. In E. Casais & C. Lewerentz (Eds.), Building Object Oriented Software Libraries Karlsruhe, Germany: FZI Publication.

[3] Chilimbi, T., Davidson, B., & Larus, J. (1999) Cache-Conscious Structure Definitions. In PLDI '99, p. 13-24.

[4] Ernst, M. D., J. Cockrell, et al. (2001). "Dynamically Discovering Likely Program Invariants to Support Program Evolution." IEEE Transactions on Software Engineering 27(2): 99-123.

[5] Fowler, M., Beck, K., Brant, J., et al. (1999). Refactoring: Improving the Design of Existing Code. Addison Wesley.

[6] Ganser, B. and R. Wille, (1999). Formal Concept Analysis - Mathematical Foundations. Springer-Verlag.

[7] Griswold, W. G. and D. Notkin (1993). "Automated Assistance for Program Restructuring." ACM Transactions on Software Engineering and Methodology 2(3): 228-269.

[8]Kataoka, Y.,Ernst, M. D.,Griswold, W. G., & Notkin, D. (2001). Automated support for program refactoring using invariants. In ICSM 2001, (pp. 736-743). Florence, Italy.

[9] Opdyke, W. F. (1991) Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. Ph.D. thesis, University of Illinois, Urbana-Champaign.

[10] Roberts, D., Brandt, J., and Johnson, R. E. (1997). A Refactoring Browser for Smalltalk. Theory and Practice of Object Systems, Vol 3, No. 4.

[11] Roberts, D. (1999) Eliminating Analysis in Refactoring. Ph.D. dissertation, University of Illinois at Urbana-Champaign.

[12] Snelting, G., & Tip, F. (2000). Reengineering Class Hierarchies Using Concept Analysis. ACM Transactions on Programming Languages and Systems, May 2000, 540-582.

[13] Strechenbach, M., & Snelting, G. (2004). Refactoring Class Hierarchies with KABA. OOPSLA '04, October 2004.

[14] Tonella, P., & Ceccato, M. (2004). Aspect Mining Through the Formal Concept Analysis of Execution Traces. In IEEE 11th Working Conference on Reverse Engineering, (pp. 112-121).

[15] Trolltech (2004) Qt cross-platform GUI library, version 3.3.2. www.trolltech.com.

[16] van Deursen, A., & Kuipers, T. (1999) Identifying Objects Using Cluster and Concept Analysis. ICSE '99, p. 246-255.

[17] Yevtushenko, S. A. (2000). System of data analysis "Concept Explorer" (in Russian.). In 7th National Conference on Artificial Intelligence KII-2000, (pp. 127-134). (Available at http://sourceforge.net/projects/conexp)