Source-to-Source Refactoring Framework for Local and Global Variables

Hemaiyer Sankaranarayanan

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Andrew Gill

Dr. Xin Fu

Date Defended

The Thesis Committee for Hemaiyer Sankaranarayanan certifies
That this is the approved version of the following thesis:

Source-to-Source Refactoring Framework for Local and Global Variables

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

To Mom and Dad

Acknowledgement

I would like to thank my advisor Dr. Prasad Kulkarni for his guidance throughout my graduate life at University of Kansas. His constant advice and suggestions have helped me many a time to find solutions to the problems encountered.

I would also like to thank Dr. Andrew Gill and Dr. Xin Fu for serving in my committee.

A special thanks to my husband, Praveen Krishnan, for supporting me during my lows and highs.

Finally, I would like to thank my family and friends for their encouragement.

Abstract

Global variables in C/C++ programs are those that are declared outside a function, and whose scope extends the lifetime of the entire program. Global variables have been shown to cause issues to program maintainability, extensibility, dependence, verification, and thread-safety. Consequently, the use of global variables has been disparaged in many programming textbooks and coding guides. However, employing global variables can also make coding more convenient and improve program performance. We have found that the use of global variables remains unabated and extensive in most real-world software programs. In this work we present our source-to-source refactoring tool to automatically detect and localize global variables in a program. Our tool implements a compiler based transformation algorithm that finds the best location to move and redefine each global variable as a local. For each global variable, our algorithm initializes the corresponding new local variable, and passes it as an argument to all functions that need it, and updates the source code line that used the global variable to now instead use the corresponding local or function argument, thus maintaining the original program semantics. In this work we further characterize the nature of how global variables are employed in common benchmark programs. We further study the effect of our transformation on static program properties, such as the change in the number of function arguments and visibility of program state. Additionally, we also quantify the effect of localizing global variables on dynamic program characteristics, including the change in *data* and *stack* memory usage and runtime program performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *Global* variable in C and C++ is one that is declared outside a function, and is in-scope and visible throughout the program. Thus, global variables are accessible and can be set and used in any program function [1]. The use of global variables has been observed to cause several problems. First, researchers have argued that global (and other *non-local*) variables increase the mental effort necessary to form an abstraction from the specific actions of a program to the effects of those actions, making it more difficult to comprehend a program that uses global variables [2]. In other words, source code is easiest to understand when we limit the scope of variables. Second, developers have found it more difficult to test and verify software that employs global variables. Use of globals makes it difficult (for humans and automatic tools) to understand the state being used and modified by a function, since globals do not need to be explicitly passed and returned from the function. Similarly, formally verifying code that uses global variables typically requires stating and proving invariant properties, which make make the verification task more arduous [3]. For such reasons, the formally-defined SPARK programming language requires the programmer to annotate all uses of

global variables [4]. Third, global variables have also been implicated in increasing *program dependence*, which measures the influence of one program component on another [5]. Additionally, global variables have also been observed to cause *dependence clusters*, where a set of program statements are all dependent on one another. A low program dependence and a lack of dependence clusters are found to benefit program comprehension [6, 7] as well as program maintenance and re-engineering [8, 9]. Fourth, the use of global variables causes the program to be *non-thread-safe* [10,11]. This is because global variables are allocated space in the data region of the process address space, providing only one copy of these variables for all program threads. Consequently, developers are generally required to eliminate global variables or synchronize their access for multi-threaded programs. Thus, on account of these limitations, the use of global variables in generally discouraged in modern programming practice.

Regardless of the problems caused by global variables, they are still extensively used in most current real-world software systems. Their use can be attributed to two (real or perceived) primary benefits of using global variables: efficiency and convenience. Researchers have shown that employing global variables can boost program efficiency and lower (stack) space usage by reducing or eliminating the overhead of argument passing and returning values during function calls/returns [12]. However, the *globalization* transformations to achieve this effect can generally be performed automatically by the compiler during the source-to-binary generation without affecting the high-level source code. It may also be more convenient for developers to hold program state that is manipulated and consumed in multiple dislocated programs regions in global variables. In such cases, it may be difficult for the programmer to determine the best place for declaring the *local*

variable and find the best path to make this variable available to all functions setting or using it. Such use of globals is especially attractive for developers updating unfamiliar code regions in large software programs. However, given the harmful effects of global variables, it will be more desirable if we could still provide developers the convenience of using global variables, but then automatically *localize* such variables to preserve the understandability, verifiability, and maintainability of the source code program.

In this work, we develop a compiler-based tool to automatically find and eliminate global variables in a program by transforming them into local variables. Our tool automatically finds the closest *dominator* function to localize each global variable, and then passes the corresponding local variable as a parameter to every function setting/using the original global. Function prototypes are appropriately modified throughout the program reflect the new parameters for each function. At the same time, each access of the global variable is updated to instead modify or use the corresponding local variable or parameter argument. In this paper we describe the algorithms we used to accomplish this transformation, and measure their effect on the space and time requirements of the modified programs. Thus, we make the following contributions in this work:

1. To our knowledge, we present the first source-to-source transformation tool to localize global variables in C programs.

2. We present detailed statistics and observations on the use of global variables in existing MiBench and SPEC benchmarks.

3. We measure and quantify the effect of our transformation on the number of function arguments passed, along with its space and size overheads.

Chapter 2

Motivation

2.1 Global Variables

Variable in languages are a portion of memory to store a determined value. A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is the one declared within the body of a function or a block.

Each local variable in a function comes into existence only when the function is called, and disappears when the function is exited. This is why such variables are usually known as automatic variables. Because automatic variables come and go with function invocation, they do not retain their values from one call to the next, and must be explicitly set upon each entry. If they are not set, they will contain garbage. A global variable in contrast is always initialized to 0. Additionally they can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

2.1.1 Disadvantages of global variables

Ever since their conception, global variables have caused almost as much controversy as the goto statement. A good programming practice is that global variables should not be overused even though they can simplify the code considerably.

"Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not all obvious - variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify" [13]. Experienced programmers support a programming philosophy called *"the principle of least privileges"*. This philosophy says that if the accessibility of a program resource, such as a function or variable, is restricted to just those portions of the program where such accessibility is absolutely required, then the programmer is less likely to introduce errors into the code. Global variables violate this principle. Avoiding the global variables have been suggested as one of the best practices for software development.

Following are some of the well-known disadvantages in using global variables in a program.

- **Increase in Code Complexity**

  A global variable has an unlimited potential for creating mutual dependencies, and adding mutual dependencies increases code complexity. Untracked interactions between different components are the archetypal defect in software engineering called *Action at a Distance* and global variables contribute to it. It happens when we run one part of the program that we believe is isolated but unexpected interactions and state changes happen in distant locations of the system. This can throw the code wildly of its track and such

bugs can be hard to track down. Ultimately it indirectly lowers developer productivity

- **Wasted Memory**

  Since global variables are alive throughout the program, its memory can't be reclaimed even if we no longer need access to it as opposed to local variables which are de-allocated when the function in which it is declared returns.

- **Non-locality**

  Source code is easiest to understand when the scope of its individual elements is limited. Global variables can be read or modified by any part of the program, making it difficult to remember or reason about every possible use. In addition it removes the logical isolation of the functions from the rest of the code.

- **Namespace Pollution**

  Since C does not have Namespace, global names are available everywhere. We may unknowingly end up using a global all the while thinking that we are using a local by misspelling or forgetting to declare the local or vice versa. Also, if we ever have to link together modules that have the same global variable names either one of the following output is expected

  1. get linking errors
  2. the linker simply treats all uses of the same name as the same object without as much as throwing a warning

  If the later is the case then external programs can overwrite user written portions of the code. As a consequence we lose some of the functionality offered and introduce subtle bugs.

- **Testing Constraints**

  It is difficult to see what variables are currently on scope of a function unless they are all passed as parameters or declared locally. Since any code anywhere in the program can change the value of the global variable at any time, understanding the use of the variable may entail understanding a large portion of the program. It also makes isolating units of code for purposes of unit testing more difficult. Thus they can directly contribute to lowering the quality of the code.

  For communicating systems, the ability to test system invariants may require running more than one copy of a system simultaneously, which is greatly hindered by any use of global variables that are not provided for sharing as part of the test.

- **Access Control Issues**

  A global variable can be get or set by any part of the program, and any rules regarding its use can be easily broken or forgotten. The lack of access control greatly hinders achieving security in situations where we may wish to run untrusted code.

- **Concurrency Issues**

  Writing code in such a way that it can be partially executed by a thread, re-executed by the same thread(Re-entrant) or simultaneously executed by another thread(thread-safe) and still correctly complete the original execution is essential for concurrency. This requires the saving of state of information in variables local to each execution, usually on a stack, instead of in global variables. If global variables have to be accessed by multiple threads of execution, synchronization is necessary. When dynamically linking modules

with global variables, the composed system might not be thread-safe even if the two independent modules tested in dozens of different contexts were safe. So data consistency may not be guaranteed. In such cases manually eliminating or synchronizing the use of globals is extremely hard, time-consuming and tedious.

Our research proposes a pure software and fully automated approach to do away with unnecessary globals in existing software applications by localizing them.

Chapter 3

Related Work

In this section we describe previous research efforts to localize global variables and techniques to manage some of the shortcomings of global variables. Many popular programming language textbooks [13] as well as several individual programming practitioners [14] have derided and discouraged the use of global variables. At the same time, acknowledging the necessaity and/or convenience of employing global variables/state, language designers have developed alternative programming constructs that provide some of the benefits while controlling many limitations of global variables. Arguably, one of the most well-known alternative to some uses of global variables is the *static* specifier in C/C++ that limits the scope of global variables to individual functions or source files [13]. Another construct that programmers often use in place global variables is the *singleton* design pattern that can encapsulate global state by restricting the instantiation of a class to a single object [15]. However, the use of the singleton pattern can result in many of the same problems with testing and code maintainence that are generally associated with global variables [16].

To our knowledge, there exist few attempts at source-to-source code refactor-

ing to automatically detect and eliminate global variables in C/C++ programs. Sward and Chamillard developed a tool to indentify global variables and add them as *locals* to the parameter list of function in Ada programs [17]. However, apart from operating only on Ada programs, this work does not describe their implementation and does not provide any results. Yang et al. proposed and implemented a "*lifting*" transformation to move global variables into `main`'s local scope [18]. *Lifting* was designed to only work with their other "*flattening*" transformation that absorbs a function into its caller without making a new copy of the function for each call-site. This earlier research aimed to place the stack allocated variables in static memory to minimize RAM usage for embeded systems applications, and did not have to deal with most of the issues encountered in a more general technique to eliminate global variables.

Most related to our current research are works that attempt to automatically eliminate global variables to generate *thread-safe* programs. Zheng et al. outlined a compiler-based approach to eliminate global variables from multi-threaded MPI (Message-Passing Interface) based Fortran programs [11]. Their transformation moves all globals into a single structure. Every MPI process gets its own instance of this structure, which is them passed as an argument to all functions. Thus, unlike our implementation, their transformation does not affect code maintainability. Additionally, this previous work also did not collect statistics on the use of global variables and the affect of the transformation on code maintainability and performance metrics. Our earlier work also implemented a similar algorithm to transform global variables into their local variants to make 'C' programs thread-safe [10]. However, this earlier work was not targeted at code maintainability and did not implement a source to source transformation. Moreover, it did not collect

and analyze the various statistics about global variables and the transformation that we present in this work.

The ultimate goal of our research is to develop a new code refactoring tool that can generally reassign storage between local and global variables. Existing code refactoring tools are typically only used to enhance non-functional aspects of the source code, including program maintainability [19] and extensibility [20]. Examples of important code refactorings for C program maintainability include renaming variables and functions, dividing code blocks into smaller chunks, and adding comments to the source codes [21]. None of the existing refactoring tools provide an ability as yet to transform global/local variables, as we perform in this work.

Chapter 4

Algorithm

4.1 Localizing Global Variables

Even moderate-sized programs in C/C++ typically contain many global variables. Additionally, these global variables may be scattered throughout the code, which make it highly tedious and error-prone to *manually* detect and refactor the code to remove these variables. Therefore, our approach employs an *automatic* compiler-driven algorithm to find and eliminate global variables. Our algorithm works by converting the global variables to locals, and then passing them as arguments to all the functions where they are needed. In this section we provide more details on our compiler-based framework and transformation algorithm.

4.1.1 Transformation Algorithm for Localizing Global Variables

Our compiler based transformation tool performs two passes to localize global variables. In the first pass, we generate the call-graph, detect global variables, and collect other information regarding the use of global variables in the program. The second pass uses this information to move global variables into the local scope

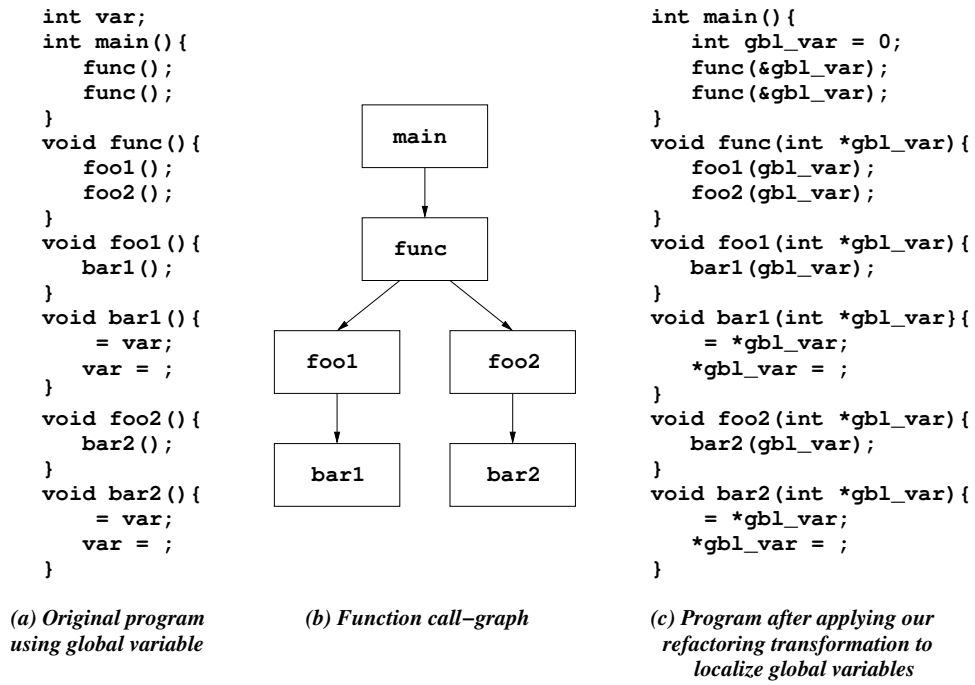

Figure 4.1. Example to illustrate the program transformation to localize global variables

of the appropriate function, pass these local variables to other functions using them, update function prototypes and the variable names in the source statements accessing each global variable. We use the small example program in Figure 4.1(a) to explain our transformation algorithm in more detail. The syntax "`= var`" in Figure 4.1(a) indicates a *use* of the variable `var`, while "`var =`" indicates a *set* of the variable `var`. The algorithm proceeds as follows.

1. In the first step we invoke the compiler to compute the static call-graph of the program. Figure 4.1(b) shows the call-graph that will be generated for the example program in Figure 4.1(a).

2. The compiler then detects all global variables in the program, as well the functions that set and/or use each global variable. We also record the data type and initialization value of each global variable.

3. Next, we automatically determine the best function to localize each global variable. While the root program function, `main()`, can act as the default localizing function for all global variables, we attempt to place each global as close as possible to the set of functions that access that variable in order to minimize the argument passing overheads. We employ our implementation of the Lengauer-Tarjan algorithm [22] to find the *immediate dominator* of each node (function) in the call-graph. A *dominator* for a control flow graph node n is defined as a node d such that every path from the entry node to n must go through d [23]. Since one global variable can be used in many functions, we further extend the Lengauer-Tarjan algorithm to find the *closest dominator* function for the set of functions that use each global variable. This closest dominator is determined by locating the first common dominator of all the functions that use that global. Thus, as an example the global variable `var` that is used in two functions, `bar1()` and `bar2()`, in Figure 4.1(a) has the function `func()` as its common dominator.

4. Simply localizing each global variable in its closest common dominator function may not retain the semantics of the original program. This is because if this dominator function is called multiple times, then it will re-declare and re-initialize the localized variable each time, which is different than the single initialization of the original global variable. Thus, in the example program in Figure 4.1(a) the closest dominator function `func()` is called multiple times from `main()`, and therefore may not be a semantically *legal* choice to locate the global variable `var`. For each global variable, we traverse the dominator tree starting from its closest common dominator up to `main()` to find the first legal dominator that is only invoked once by the

program.

5. Next, our transformation moves each global variable as a local variable to its closest legal dominator function. The transformation also adds new instructions to this function to correctly initialize the new local variable. In Figure 4.1(a), the global variable `var` is moved to the function `main()` and initialized as the new local variable `glob_var`.

6. The next step involves finding the functions in the call-graph between the legal dominator and all the functions where the global is used. We call this set of functions as the global variable's *frontier*. The local copy for each global variable needs to be passed by reference to each of its frontier functions in order to reach their appropriate end locations, where they are used. This requires modifying the calling interface of each frontier function. Thus, in program 4.1(a), the local variable `glob_var` is passed by reference to all its frontier functions, namely `func()`, `foo1()` and `foo2()`.

7. The final step in our transformation is to modify the calling interface of the end functions for each global variable to get the additional arguments corresponding to the local variants of global variables. Thus, the calling interface of functions `bar1()` and `bar2()` is updated to accept the address of the local variable `glob_var` as an argument. Our tool then automatically updates every use of each global variable to instead use its corresponding local variant.

Thus, our algorithm to eliminate global variables will automatically transform the program in Figure 4.1(a) to the program in Figure 4.1(c). Our tool has the ability to either transform all possible global variables in the program, or to

selectively apply the transformation to individual globals that are specified by the user.

Chapter 5

Compiler Framework

We have implemented our algorithm to localize global variables as a source-to-source transformation using the Clang compiler framework. In this section we first describe our compiler framework along with the limitations that were imposed by the framework on our algorithm implementation.

5.1 Clang

We use the modern and popular Clang/LLVM [24, 25] compiler for this work. LLVM provides a mature SSA-based compiler *backend* [23] that supports both static and dynamic compilation and optimizations for programs written in different languages and across multiple target architectures. Clang is a modern C/C++/Objective-C *frontend* for LLVM that provides fast code transformation and useful error detection and handling ability. Clang also exposes an extensive library of functions that can be used to build tools to parse and transform source code. In this project we employ the extensive Clang library to build a source-to-source transformation tool for localizing global variables.

libbasic	Diagnostics, SourceLocations, SourceBuffer abstraction, file system caching for input source files
libast	Provides classes to represent the C AST, the C type system, built-in functions, and various helpers for analyzing and manipulating the AST
liblex	Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion
libparse	Parsing, invokes coarse-grained Actions provided by the client but knows nothing about ASTs or other client specific data structures
libsema	Semantic Analysis, provides a set of parser actions to build a standardized AST for programs
librewrite	Editing of text buffer for code rewriting transformation, like refactoring
libanalysis	Static analysis support
libindex	Cross-translation-unit infrastructure and indexing support
driver	A driver program, client of the libraries at various levels

Table 5.1. Clang Libraries

5.1.1 Clang as Rewriter

Clang was designed to retain more information during the compilation process than GCC, and preserve the overall form of the original code. This makes it easier to map errors back into the original source. The parse tree built by clang is more suitable for supporting automated code refactoring as it remains in a parseable text form at all times. Clang is highly modularized, based almost entirely on replaceable link-time libraries as opposed to source-code modules that are combined at compile. A library-based architecture makes the reuse and integration of functionality provided by clang more flexible and easier to integrate into other projects. The table shows clang's base libraries.

Clang's FrontendAction is the task that can be performed on an built Abstract Syntax Tree(AST). AST is a tree representation of the abstract syntactic structure of the source code written in a high-level programming language. Each node of the tree denotes a construct occurring in the source code. Clang has only one

type of an AST for C based languages like C++, C and ObjectiveC. Unlike most common compilers, Clang's AST preserves lots of high-level information, which is highly useful for doing the refactoring.It implements the AST visitor interface called ASTConsumer. Using this interface and the rewriter library we perform the necessary analysis on the source code and make changes accordingly.

Using the rewriter library's API changes to the source can be made selectively, which are then mirrored in the original source code. The rewriter has objects that store the entire source buffer. It also provides interface to modify this buffer. As code is rewritten, source buffer from the original input with modifications get a new Rewrite Buffer associated with them. The Rewrite Buffer captures modified text itself as information used to map between Source locations in the original input and offsets in Rewrite Buffer. Finally this modifier buffer is redirected to an output file.

Chapter 6

Implementation

In this chapter we describe the process of localizing the global variables in C programs. This is a completely generalized approach and experimental results show that it is practical on real-world examples. We built a plugin that is dynamically linked to clang compiler tree at runtime. The input to the tool is a C program and output is the transformed C program with localized global variables.

To analyze a program, our system first parses the source file and constructs a single AST in memory. Then we collect all the definitions and uses of all global variables in the program by traversing the AST. For every global variable we collect the set of functions in which it is used. We find the best function to which the definition of the global can be moved. This function is called the global dominator. Additionally we find the set of frontier functions for each global variable. The storage for the global variable is then moved from the global data storage into the local data storage in its dominator function. The global then is passed by reference to all the function in the global variable's frontier.

We will see each of the above steps in detail in the Phases section below.

6.1 Phases

6.1.1 Setup

- **Source File Setup**

  For every benchmark program we check whether all the header files have defined the header guards and define one if not. This makes sure that header file contents are not included more than once inadvertently in the including source file. Then we concatenate all the source files (.c) to make a single .c file. ASTContext object used heavily in our refactoring tool is unique for every source file. AST Objects associated with it are released when the ASTContext is itself destroyed i.e. when the compiler moves to next source file. Usage and definition of some of the functions and global variables span multiple files and their objects are required to be present in memory throughout our analysis and transformation phase. The concatenated file is then preprocessed so as to expand any macros present. Since clang's rewriter API does not support macro rewriting this step is indispensable.

- **Static and Dynamic Call Graph Generation**

  The static call graph is then generated using clang. The output is a dot file, which along with the dynamic call information would be then read by the refactoring tool along with the source file that needs to be transformed. After a single run of the original benchmark program the dynamic call graph is generated. This is done using user written profiler extension for GNU gcc compiler.

6.1.2 Analysis

- **Dominance Analysis** First we find the dominator for each function in the program. This will further help in pruning call graph orphans and finding functions involved in loops. We then compute the global variable dominator to which the declaration of the variable is moved in the transformation phase. Global frontier functions are also deduced to which the global variable will be later in transformation passed as an argument.

- **Global Variable Analysis** In this phase we classify all the global variables to Read-Only(RWO), Read-Write(RW) and Unused categories. Read-only variables contain some constant information, and usually they reflect what is known at the time of the allocation. The read-only variables are used mainly for comparison operations and their values are never modified. Write-Only variables are just written to but their value is never used. Both these class of variables form the RWO class. From maintenance point of view, we avoid localizing them, as we cannot justify the time and memory overhead. We also let the unused variable as it is. The RW variable has an initial value that is overridden through the course of the program. This section of the global variables is up for localization.

  At the end of analysis phase we will have detailed information about all the global variables and we proceed with localizing relevant global variables.

6.1.3 Transformation

- **Rewrite Declarations** In this phase we remove the original global declaration and move it to its dominator function. The name of the variable is modified to include the term global in the name so as to not collide with

any existing local variable names with the same name. Also since the loader before program execution always initializes global variables, we also have to explicitly initialize it. Straightforward initialization is done if it is scalar or if it is a vector then it is initialized with memset library function.

- **Rewrite Function Prototypes** For every function we also have computed the list of global variables that are to be passed to it so we update the function prototype with this additional set of global variables. This is done by traversing the AST for function prototypes and modifying it as we encounter it.

- **Rewrite Call Expressions** We also traverse the call expression and update them with the added global variables for each function. Global variables are always passed as reference to retain any changes made to it.

- **Rewrite Statements** For every function we have built a small table, which maps list of original global variable name and the new name. We perform analysis to see if the function has any local variable that has the same name as the global variable. Since the local in the particular function overrides the global variable we make sure not to modify function statements that may be actually just referring to its local variable. This phase is actually done using the user written lexer.

6.2 Limitations Imposed by the Current Compilation Framework

In this section we describe some of the limitations in our current algorithm implementation that are present due to some of the restrictions in our selected

compiler framework. We note that none of these shortcomings are fundamental restrictions on the algorithm, but we may need better support from the underlying compiler to resolve these issues in the future.

6.2.1 Precise Call-Graph Generation in the Presence of Function Pointers

Our compiler-based static approach to localize global variables requires the construction of a precise call-graph for each program. Unfortunately, the presence of indirect function calls via function pointers makes precise call-graph construction difficult in languages like C/C++ [26, 27]. We also find that most larger benchmarks make generous use of indirect function calls. The Clang/LLVM compiler that we use for this project does not yet perform precise pointer analysis necessary for the proper resolution of all function pointers and indirect function calls. This limitation can produce incomplete call-graphs, which may later cause our transformation to generate code that is semantically inconsistent with the original program.

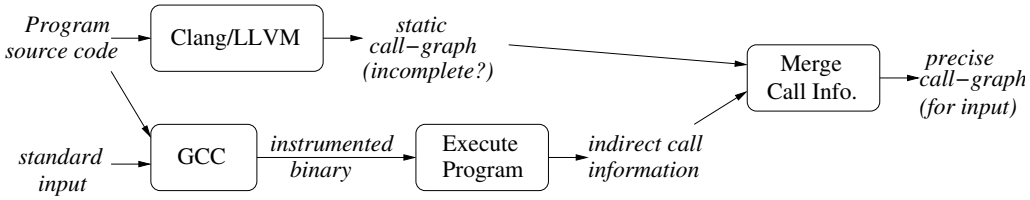

Figure 6.1. Framework for obtaining precise call-graph information for our analysis experiments

In our current work we circumvent this problem of incomplete call-graphs by supplementing the compiler-generated static call-graph with profiling-based information linking indirect function call-sites with their targets for each benchmark-input pair. Our framework for call-graph generation is illustrated in Figure 6.1.

We use the Clang/LLVM static compiler to generate the (possibly incomplete) static call-graph. We then employ our modified variant of GCC (version 4.5.2) [28] to instrument each source file with additional instructions that produce the (*caller* $\rightarrow$ *callee*) function relationships at every indirect call on program execution. We modified GCC to provide this instrumentation ability especially for our current research. We then use this supplemental indirect function call information to complete the static call-graph, if necessary. We note that the use of function pointers is not a limitation of our general technique, since precise function pointer analysis and call-graph construction has been shown to be feasible for most programs in earlier studies [29]. Thus, our workaround is intended to only provide a temporary fix until the ability of generating precise call-graphs is integrated into the Clang compilation environment.

6.2.2 Incomplete Variable Name Alias Analysis

Another related concern in C/C++ programs is correctly dealing with variable *aliasing*. Aliasing occurs when a data location in memory can be accessed through different symbolic names in the program. Along with complete pointer analysis, precise alias analysis to locate aliased variables that point to the same memory address is also critical to accurately localize all global variables for all programs. Precise alias analysis is also not currently implemented in the Clang source-to-source compiler. Our transformation tool is able to detect simple aliasing cases when a global variable is passed as a parameter (with or without a distinct name) to another function. In such cases, our tool automatically links the global variable name with the parameter name in that function. However, we do not yet handle more complex aliasing cases in the source codes.

6.2.3 Support for Multiple Source Files

A final limitation of our present tool setup is manifested due to the current implementation of the Clang compiler frontend that only allow it to hold the parse tree representation and other internal data structures for one (or the current) file at a time. Thus, Clang destroys the previous AST (Abstract Syntax Tree) representation before reading each next file from the command-line. This limitation does not prevent the generation of a static call-graph over multiple source files since all information to produce the call-graph is maintained in external data structures. However, the transformation for localizing global variables may require updates to multiple function ASTs across different source files simultaneously. Such inter-file AST updates are currently not possible with the Clang compiler.

We have not yet attempted to modify this default behavior of the Clang compiler to handle multiple 'C' program source files on the command-line. Instead, for our current project we perform a simple pre-processing pass on multi-file input programs to concatenate the multiple input files into a single file that is then handed to the Clang compiler. While this solution works for most of our multi-file benchmark programs, it fails for a few applications that employ file-level statics with the same names in multiple different files.

6.2.4 Incorrect Static Call-Graph

We also find that our version of the Clang compiler sometimes generates an incomplete static call-graph even for the set of functions that contain no function pointers or indirect calls. This is a separate problem than the issue of imprecise call-graph generation due to indirect calls, and therefore we decided to not extend

our workaround from Section 6.2.1 to cover all program function calls. Instead, we are currently working to understand and resolve this problem in the Clang compiler toolset.

The Clang/LLVM framework is a rapidly evolving target with several separate groups resolving existing issues and adding novel features to the toolset. We will work with these open-source groups to extend the Clang compiler and resolve some of the issues discussed in this section. Moreover, we note that none of these issues pose a fundamental restriction on the benefit of our tool to eliminate local variables for most programs.

Chapter 7

Benchmark Framework

We have collected a rich and extensive set of benchmark programs to analyze the use of global variables in existing programs and validate the behavior of our transformation tool to eliminate global variables. Our benchmark set includes 14 benchmarks from the MiBench suite [30] and five benchmarks from SPEC CPU CINT2006 benchmark suite [31]. The MiBench benchmarks include popular C applications targeting specific areas of the embedded market. The standard SPEC suite allows us to experiment with larger and more complex general-purpose applications. In addition, the following MiBench benchmarks were analyzed but not included in our experimental set since they do not contain any global variables: *basicmath*, *crc32*, *fft*, *patricia*, *qsort*, *rijndael*, *sha*, and *susan*.

Unfortunately, issues caused by the lack of multi-file support and incomplete static call-graph generation discussed in the last section prevent the proper handling of some of the larger benchmarks by our current tool implementation. As a result, we leave out the some MiBench (*lame*, *typeset*, *ghostscript*, *sphinx*) and SPEC (*400.perlbench*, *403.gcc*, *445.gobmk*, *464.h264ref*) benchmarks from our experimental set. Additionally, *rsynth* from MiBench was not included as it produces

Benchmark	LOC	Func	Description
MiBench benchmarks			
adpcm	114	8	compress 16-bit linear PCM samples to 4-bit
bitcount		33	test processor bit manipulation abilities
blowfish	299	17	symmetric block cipher with variable length key
dijkstra	70	15	Dijkstra's shortest path algorithm
gsm		76	GSM voice encoding/decoding algorithm
ispell	2,802	167	fast spelling checker
jpeg	1,256	408	image compression and decompression
mad		217	high-quality MPEG audio decoder
pgp		454	public key encryption algorithm
stringsearch	134	20	searches for given words in phrases
tiff2bw		409	convert color *tiff* image to b&w image
tiff2rgba		406	convert color *tiff* image to RGB *tiff* image
tiffdither		399	reduce image resolution/size at expense of clarity
tiffmedian		412	convert image to a reduced color palette
SPEC CINT benchmarks			
401.bzip2	2,852	117	popular compression program v1.0.3
429.mcf	699	38	network simple algorithm for vehicle scheduling
456.hmmer	12,039	592	protein sequence analysis using Markov models
458.sjeng	4,935	174	a highly-ranked chess program
462.libquantum		125	simulate factorization algo. on quantum comp.

Table 7.1. Our set of benchmark programs (LOC – counts the number of lines containing a semi-colon; Func – counts the number of static function definitions in each program).

no traceable output to verify the correctness of our transformation. In spite of these restrictions, we believe that our benchmark set is large and diverse enough to allow a good understanding and generalization of the properties of our algorithm implementation. Table 7.1 contains descriptions of our selected benchmark programs.

Table 7.2 shows the static characteristics of global variables in our benchmark programs. We only show those benchmarks that have at least one global variable. For each benchmark listed in the first column, the remaining columns successively show the total number of global variables declared in the program (*total*), the number of read-only or write-only global variables (*RO/WO*), the number of un-

Benchmark	Total	RO/WO	Unused	RW	Moved
adpcm	5	4	0	1	1
bitcount	1	0	0	1	1
blowfish	2	0	1	1	1
dijkstra	10	0	0	10	10
gsm	22	1	3	18	6
ispell	97	5	14	78	69
jpeg	15	5	7	3	3
mad	38	5	24	9	2
pgp	276	63	11	202	147
rijndael	8	1	2	5	5
stringsearch	8	0	5	3	3
tiff2bw	44	10	24	10	9
tiff2rgba	36	8	23	5	3
tiffdither	39	12	14	13	7
tiffmedian	51	7	25	19	17
401.bzip2	30	9	13	8	8
429.mcf	8	1	0	7	7
456.hmmer	48	26	7	15	7
458.sjeng	244	45	23	176	166
462.libquantum	10	0	0	10	8

Table 7.2. Number and type of global variables in benchmark programs.

used global variables (*Unused*), and the number of globals that are both read as well as written by the program (*RW*). Our transformation algorithm only considers the variables in the *RW* category as potential candidates from moving as local variables.

The final column in Table 7.2 shows the number of *RW* global variables that were successfully localized by our transformation algorithm for each benchmark. Thus, we can see that while our tool is able to localize most global variables, it fails in a small number of cases. We have categorized these failed cases into three primary sets: (a) Global variables used in calls to the `sizeof` function: After our transformation these calls fail to provide the correct size for the corresponding function parameters that are passed via reference. (b) Global variables used in

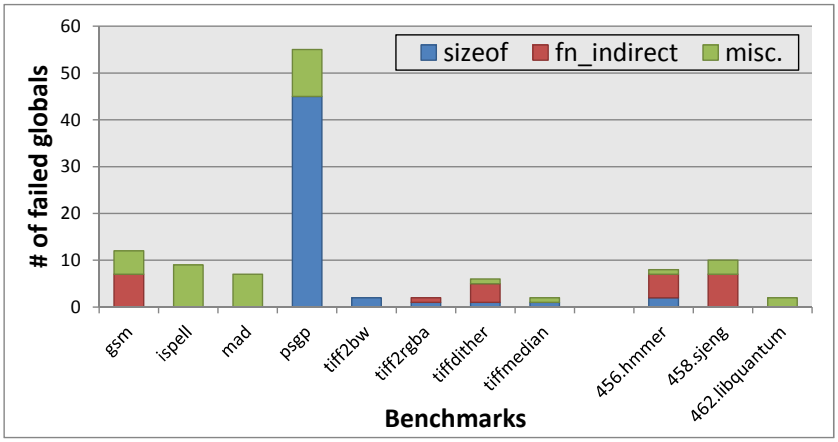

Figure 7.1. Categories and number of global variables that our tool fails localize for each benchmark

functions called indirectly: We do not yet update function pointer declarations. (c) Miscellaneous: Global variables that cause the compiler to generate incorrect code, if transformed. We studied a few of these cases, and found most of them to occur due to the imprecise alias analysis performed by the Clang compiler. Global variables belonging to the first two sets are automatically detected and bypassed by our tool, with a message sent to the user. Figure 7.1 plots the number of global variables in each of these categories for every benchmark for which our tool is not able to eliminate all *RW* global variables.

7.1 Properties of Global Variables in Benchmark Programs

It is believed that developers typically employ global variables as a convenience feature when a particular program state is set or accessed in multiple program functions, and it is difficult to determine the best place to declare the variable so it can be made visible to all program regions that need the variable. Global variables are also sometimes used to improve program efficiency by reducing the

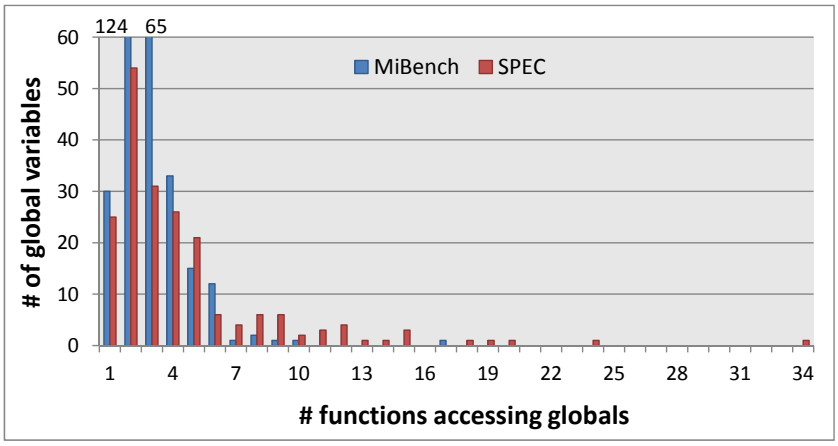

Figure 7.2. Number of functions accessing global variables

overhead of passing the variable to several different program functions. Figure 7.2 plots the number of functions that use/set each global variable. Thus, the first set of bars in Figure 7.2 shows that 30 global variables in the MiBench benchmarks, and 25 global variables in our set of SPEC benchmarks are only accessed by one function in the program. We uniformly accumulate all global variables from each of our benchmark suites for this plot. Thus, we can see from this figure that most global variables are only used in a small number of program functions. We reason that such usage trends indicate either poor programming practices or scenarios where the developer may not be comfortable with a large program code base. We believe that our automatic source-to-source transformation tool to localize globals will be very useful to resolve such improper uses of global variables.

Chapter 8

Results

In this section we describe our results and observations that characterize the properties of our transformation to eliminate global variables. Our experiments employ the set of standard benchmark programs described in Section **??** to determine properties regarding the use of global variables in typical C programs, and static (source code visible) and dynamic (performance) effects of our localizing transformation.

8.1 Static Characteristics of Our Transformation Algorithm

Our transformation to eliminate global variables can affect many aspects of the static program characteristics. In this section we quantify and analyze some effects of our transformation on static program properties. Our experiments in this section use the algorithm described in Section 4.1 to localize all the global variables in the *Moved* column of Table 7.2.

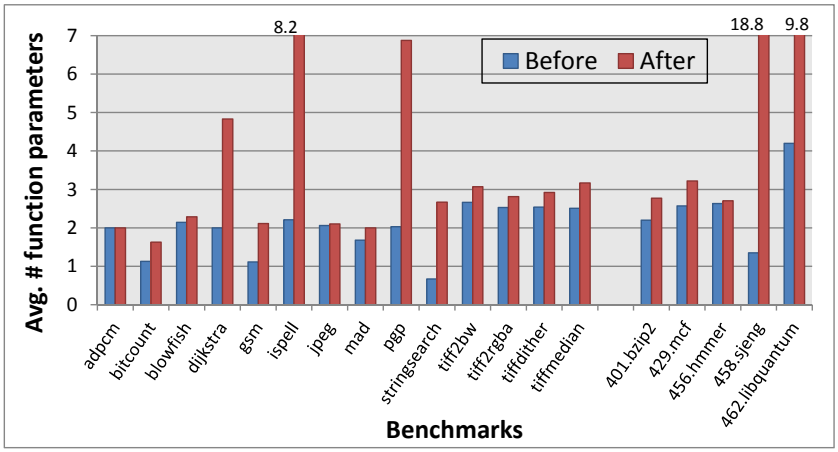

Figure 8.1. Average number of function parameters before and after applying our localizing transformation to eliminate global variables

8.1.1 Effect on Average and Maximum Function Arguments

After localizing the global variables, our algorithm needs to make their values available to all functions that set/used the original global variable. We make the new local variable accessible by explicitly passing it as an argument to all functions that need it. This scheme adds additional parameters to several function declarations in the transformed program. Figures 8.1 and 8.2 respectively plot the average and maximum number of function parameters over all the functions in each of our benchmark programs.

Thus, we can see that the change in the average and maximum number of function arguments does not change a lot for most of the benchmark programs, although it can change significantly for some programs. On average, we find that the average number of function arguments increases from 1.95 to 3.33 for MiBench benchmarks and from 2.59 to 7.45 for SPEC programs. Similarly, the maximum number of function arguments increase, on average, from 6.78 to 16.50 for MiBench programs and from 10.4 to 40.4 for SPEC benchmarks. An important and desirable side-effect of our transformation is to reveal the declarations

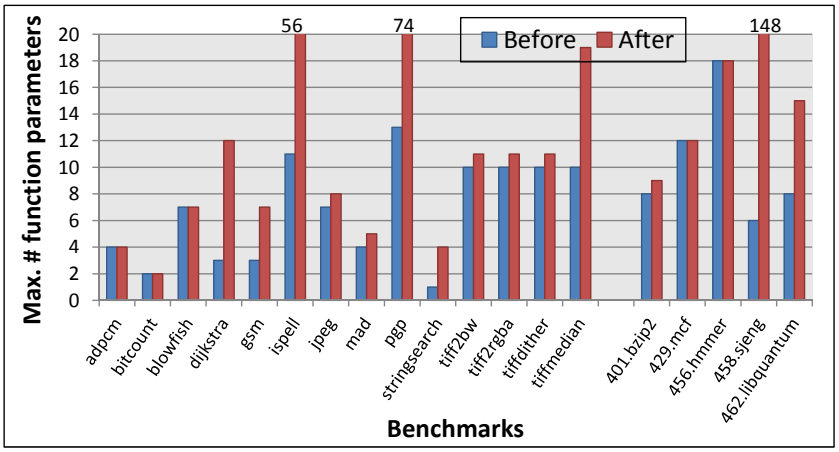

Figure 8.2. Maximum number of function parameters before and after applying our localizing transformation to eliminate global variables

of all variables used/set in any function in that function itself. This property is particularly important both from the aspects of program maintainability and verifiability. Unfortunately, passing additional function arguments can have an adverse effect of program efficiency, and we will explore the dynamic performance properties of our transformation in a later section.

8.1.2 Number of Frontier Functions

In order to make each new local variable available in all the functions that used/set the corresponding global variable in the original program, we may need to pass the local as an argument to intermediate (or *frontier*) functions that do not themselves use the local variable apart from sending them to other functions (see Figure 4.1). Figure 8.3 presents the number of frontier functions for every transformed variable. The first two bars in Figure 8.3 reveal that 12 of the new local variables in the MiBench benchmarks and one new local variable in the SPEC benchmarks have zero frontier functions. Thus, we can see that most local variables have only a small number of frontier functions. This observations shows

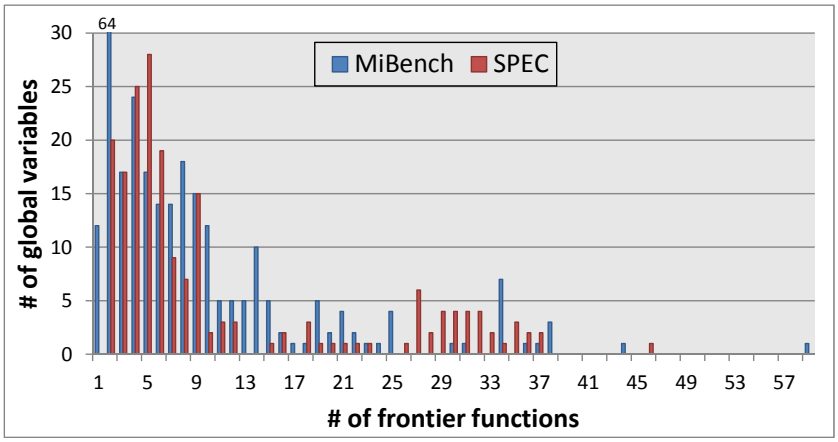

Figure 8.3. Number of frontier functions needed for the transformed local variables

that most global variables are used in functions that are located close to each other in the static program call-graph. However, many global variables are used in functions that are considerably dislocated in the program call-graph. Thus, at the other extreme, we find that there is one function each in MiBench and SPEC that has 58 and 45 frontier functions respectively. Global variables employed in such dislocated call-graph functions will likely require more user effort to *manually* eliminate, and also seem to be more sensible scenarios for the developer using global variables. By automatically handling such variables, our tool can provide the programmer the convenience of using global variables in such difficult situations, but eliminate them later to satisfy software engineering goals.

8.1.3 Effect on Program State Visibility

Global variables are visible and accessible to all functions in the program, which is argued to make if more difficult for automatic program verification and program maintainability. One goal of our localizing transformation is to reduce the visibility of all variables to only the program regions where they are needed,

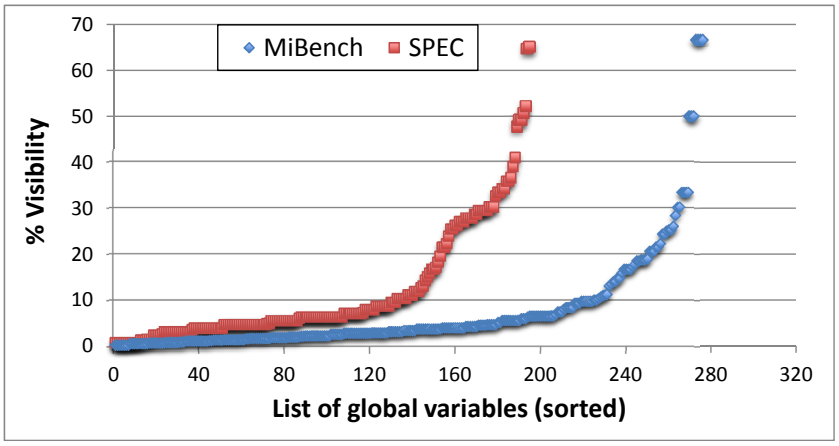

Figure 8.4. Percentage reduction in the visibility of transformed variables as compared to globals that are visible throughout the program

and assist and program verification and maintainability. Figure 8.4 plots the percentage reduction in the visibility of the each transformed local variable. There is a point-plot for each transformed variable in Figure 8.4, sorted by its percentage visibility for all MiBench and SPEC benchmark variables. This figure shows that program visibility is drastically reduced for almost (originally) global variables after transformation. For example over 81% of the global variables in MiBench programs and 68% of variables in SPEC benchmarks are visible in less than 10% of their respective program after the transformation. How about average visibility per benchmark ? (Smaller benchmarks will have higher percentage visibility?).

8.2 Dynamic Characteristics of Our Transformation Algorithm

At a lower level the transformation algorithm to eliminate global variables can have the following effects on memory consumption and program performance.

- Localizing global variables will move them out of the *data* region to the

respective function activation records (or the *stack* region) of the process address space. This movement may reduce the size of the data region, but will supplement this reduction with a corresponding increase in the size of the stack. [1]

- Each localized variable may need to be passed as an argument to other functions that access it. This operation may increase the function call overhead, as well as increase the size on the function activation record (stack).

- While global variables are either initialized statically or implicitly by the operating system, after localization, the corresponding local variables will need to be explicitly initialized in the program by the compiler. This initialization may be a source of additional overhead at runtime.

In this section we present results that quantify the memory space and runtime performance of the program before and after our transformation. For all these experiments all our benchmark programs were compiled with GCC (version 4.5.2) [28] with the '-O2' optimization flag. [2] We also built a simple GCC-based instrumentation framework to enable us to measure the *maximum* stack space requirement and program dynamic instruction counts for each benchmark. This framework is described in the next section. The MiBench and SPEC benchmarks were run with their *small* and *test* inputs respectively. The outputs produced by each program with and without our transformation were compared to validate the correctness of our tool.

[1] Assuming that each global is moved to a function that is only called once.

[2] The original *tiff2rgba* benchmark program failed to run correctly with GCC's -O2 optimizations. Therefore, this program was run unoptimized with -O0 flag.

8.2.1 GCC-Based Instrumentation Framework

We updated the GCC compiler to instrument the program during code generation. Our instrumentations can generate two types of execution *profiles* at program runtime. (a) One set of instrumentations output the *stack pointer* register on every function entry, after it sets up its activation record. The difference between the minimum and maximum stack pointer values gives us the maximum extent of the stack for that particular program run. (b) Our other set of instrumentations are added to the start of every basic block to produce a linear trace of the basic blocks reached during execution. We also modified GCC to generate a file during compilation that contains a list of all program basic blocks along with their set of instructions. The knowledge of the blocks that are reached at runtime and the number of instructions in each block allow us to compute the *dynamic instruction counts* for a particular program run. Since our instrumentations only modify the compiled benchmark code, this added code can only count the instructions in the application program and not in the library functions. We believe that dynamic instruction counts are a good supplement to actual program run-times since they are deterministic and cannot be affected by any hardware and operating system affects.

8.2.2 Effect on Maximum Stack and Data Size

For most existing systems, global variables reside in the *data* region of the process address space, while local variables and function arguments reside in the function activation record on the process *stack*. Therefore, our transformation to convert global variables into locals (that are passed around as additional function arguments) have the potential of reducing the *data* region space and expanding the

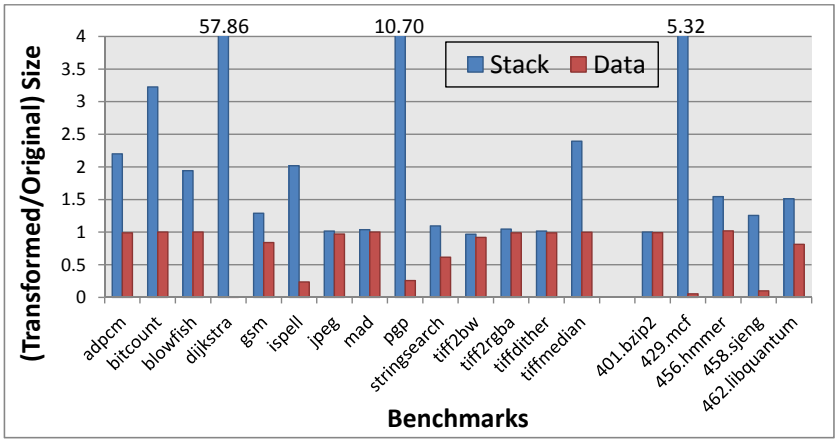

Figure 8.5. Ratio of the total *data* and maximum *stack* area consumed by each process at runtime before and after the localizing transformation

process *stack*. We use our GCC based stack-pointer instrumentation to gather the maximum required stack space (in bytes) for each benchmark run with its standard input. We also employ the Linux *size* tool to determine the space occupied by the data region of each program. Figure 8.5 plots the ratio of the total *data* and maximum *stack* requirement for each of our benchmark programs before and after the transformation to eliminate global variables.

Thus, we can see that our transformation increases the stack requirement while reducing the data space size for most programs. While some benchmarks, including *dijkstra*, *pgp* and *429.mcf*, may experience a large increase in maximum stack usage, many of these also notice a correlating reduction in the data region size. At the same, it is important to realize that while a program only maintains one copy of any global variable, multiple copies of the corresponding local variable may reside simultaneously on the stack for the transformed program. Therefore, there exist program, such as *adpcm*, *bitcount*, and *blowfish*, that show no discernible reduction in *data* size, but still encounter significant increases in maximum stack space use.

8.2.3 Effect on Dynamic Performance

Is this section we present results on experiments that quantify the effect of eliminating global variables on program performance. We employ two metrics for performance estimation. First, we use the GCC instrumentation framework to measure each program's dynamic instruction count before and after applying our transformation. Dynamic instruction counts can provide a good and deterministic estimation of actual program performance, but cannot account for differing instruction latencies, and variations due to memory, cache, and other micro-architectural effects. Second, we execute each benchmark natively on a dedicated x86-Linux machine to gather actual program *run-time*. Each benchmark is run in isolation to prevent interference from other user programs. To account for inherent timing variations during the benchmark runs, all the performance results in this paper report the average over 15 runs for each benchmark.

Figure 8.6 shows the results of these performance experiments. For the actual program run-times, we employ a statistically rigorous evaluation methodology, and only present results that show a statistically significant performance effect (with a 95% confidence interval) [32]. Thus, we can see that the localizing transformation does not produce a large performance overhead for most benchmarks. The dynamic instruction counts for most benchmarks with a small number of *Moved* global variables do not undergo a substantial change. However, the dynamic instruction counts do show large degradations in cases where the transformation localizes a large number of global variables and/or significantly increases the number of function arguments (as seen in Figure 8.1). Several benchmark program including such as *dijkstra*, *ispell*, *stringsearch*, and *458.sjeng* fall into this category. Interesting, we observe that, in most cases, the increases in dynamic instruction

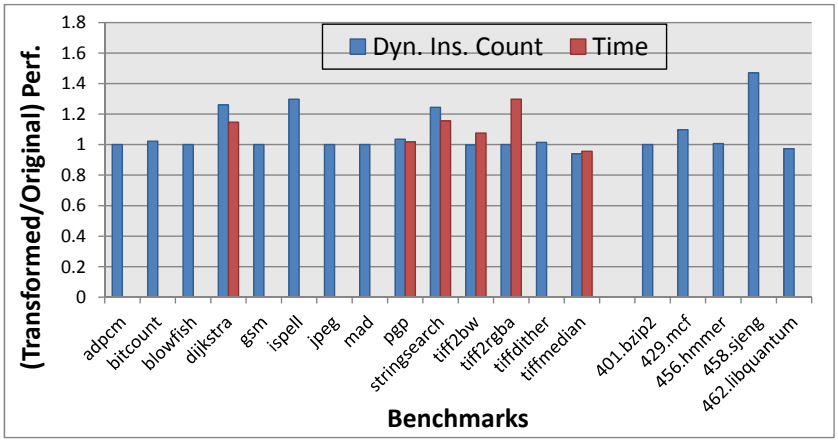

Figure 8.6. Ratio of the *dynamic instruction counts* and actual program *run-time* before and after the localizing transformation

count do not produce a corresponding increase in the actual benchmark runtime. This result may be due to the inherent inaccuracy in our instrumentation framework that only updates application code (and not library code), or due to GCC's -O2 optimization that do a good job of negating the overhead of our localizing transformation.[3]

[3]The run-time for *tiff2rgba*, the only program not compiled with -O2 optimizations, degrades substantially over the original program run-time.

Chapter 9

Future Work

There are a number of improvements that can be performed to address the limitations in our existing compiler framework, and increase the performance and attractiveness of the presented transformation. First, we plan to implement more precise pointer analysis and improve alias analysis in the Clang/LLVM compiler. Pointer analysis is necessary to appropriately resolve indirect function calls and build a precise call-graph for each benchmark. More accurate alias analysis will allow the transformation to precisely detect and merge all aliased variable names, and make our tool more robust. Second, we will extend the Clang compiler to enable it to maintain the internal AST data structures across files. This capability will allow our tool to analyze and transform application programs consisting of multiple source files. Third, we plan to explore techniques to move each global variable closer to the functions where it is needed. Currently, we employ the Lengauer-Tarjan algorithm to find the *closest dominator* function, but are often not able to localize the global variable in that function if it is called in a loop or from multiple call-sites. However, a global variable can be located in such a function if the localized variable is always set before being used. We will stati-

cally analyze the function control-flow graph to detect such cases to allow keeping the localized declaration closer to its region of use to minimize the number of frontier functions and the argument passing overhead. Third, we will further investigate the causes of performance overhead and develop optimizations to reduce this overhead of our transformation tool. Optimizations may include techniques to combine the initialization of different localized variables, and to reduce the overhead of argument passing during code generation in the compiler backend. Fourth, we plan to improve the user interface to provide users with a more intuitive coding and maintenance experience. Finally, we require good metrics to evaluate the benefit of our tool during program development, maintenance, verification, and thread-safety. We plan to study these topics and attempt to develop such metrics in the future years.

Chapter 10

Conclusion

In this paper we present our compiler-based source-to-source transformation packaged into a refactoring tool to automatically transform global variables into locals. Our transformation algorithm automatically detects global variables and where they are used. For each global variable, the tool has the ability to find the best place to redefine it as a local, appropriately initialize it, pass it as an argument to all the functions that set/use it, and then modify all program statements that used that global variable to now instead use the corresponding local or function argument. Our compiler based transformation tool is implemented using the popular, standard, and rapidly evolving Clang/LLVM compiler framework. We also analyzed the static and runtime effects of our localizing transformation to enable the developer in making an informed decision regarding whether to localize any/all global variables. We found that most of our benchmark functions make generous use of global variables. However, most of these globals are only used in a very small number of program functions that are located close to each other in the function call-graph. Therefore, localizing such global variables greatly minimizes the percentage visibility of global program state, which can assist code

verification efforts. At the same time our transformation can significantly affect the amount and distribution of memory space consumed by the *data* and *stack* regions of the process address space. Additionally, we also found that localizing most global variables only has a minor degrading effect on runtime performance, if any.

Bibliography

[1] B. W. Kernighan, D. M. Ritchie, The C programming language, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[2] W. Wulf, M. Shaw, Global variable considered harmful, SIGPLAN Not. 8 (1973) 28–34. doi:http://doi.acm.org/10.1145/953353.953355.
URL http://doi.acm.org/10.1145/953353.953355

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood, sel4: formal verification of an os kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, ACM, New York, NY, USA, 2009, pp. 207–220. doi:http://doi.acm.org/10.1145/1629575.1629596.
URL http://doi.acm.org/10.1145/1629575.1629596

[4] J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] D. Binkley, M. Harman, Y. Hassoun, S. Islam, Z. Li, Assessing the impact of global variables on program dependence and dependence clusters, J. Syst.

Softw. 83 (2010) 96–107. doi:10.1016/j.jss.2009.03.038.
URL http://dl.acm.org/citation.cfm?id=1663656.1663918

[6] F. Balmas, Using dependence graphs as a support to document programs, in: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 145–154.
URL http://dl.acm.org/citation.cfm?id=827253.827735

[7] Y. Deng, S. Kothari, Y. Namara, Program slice browser, in: Proceedings of the 9th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2001, pp. 50–59.
URL http://dl.acm.org/citation.cfm?id=876902.881283

[8] S. Black, Computing ripple effect for software maintenance, Journal of Software Maintenance 13 (2001) 263–279.
URL http://dl.acm.org/citation.cfm?id=511193.511196

[9] P. Tonella, Using a concept lattice of decomposition slices for program understanding and impact analysis, IEEE Trans. Softw. Eng. 29 (2003) 495–509. doi:10.1109/TSE.2003.1205178.
URL http://dl.acm.org/citation.cfm?id=1435631.859043

[10] A. R. Smith, P. A. Kulkarni, Localizing globals and statics to make c programs thread-safe, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, ACM, New York, NY, USA, 2011, pp. 205–214. doi:http://doi.acm.org/10.1145/2038698.2038730.
URL http://doi.acm.org/10.1145/2038698.2038730

[11] G. Zheng, S. Negara, C. L. Mendes, L. V. Kale, E. R. Rodrigues, Automatic handling of global variables for multi-threaded mpi programs, in: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on, 2011, pp. 220–227. doi:10.1109/ICPADS.2011.33.

[12] P. Sestoft, Replacing function parameters by global variables, in: Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA '89, ACM, New York, NY, USA, 1989, pp. 39–53. doi:http://doi.acm.org/10.1145/99370.99374.
URL http://doi.acm.org/10.1145/99370.99374

[13] B. W. Kernighan, The C Programming Language, 2nd Edition, Prentice Hall Professional Technical Reference, 1988.

[14] c2.com Wiki contributors, Global variables are bad, http://c2.com/cgi/wiki?GlobalVariablesAreBad (November 2011 (last accessed)).

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] M. Hevery, Clean code talks - global state and singletons, Google Tech Talks (November 2008).

[17] R. E. Sward, A. T. Chamillard, Re-engineering global variables in ada, in: Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies, SIGAda '04, ACM,

New York, NY, USA, 2004, pp. 29–34. doi:http://doi.acm.org/10.1145/1032297.1032303.
URL http://doi.acm.org/10.1145/1032297.1032303

[18] X. Yang, N. Cooprider, J. Regehr, Eliminating the call stack to save ram, in: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '09, ACM, New York, NY, USA, 2009, pp. 60–69. doi:http://doi.acm.org/10.1145/1542452.1542461.
URL http://doi.acm.org/10.1145/1542452.1542461

[19] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 1st Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[20] J. Kerievsky, Refactoring to Patterns, Pearson Higher Education, 2004.

[21] D. Wilking, U. F. Khan, S. Kowalewski, An empirical evaluation of refactoring, e-Informatica Software Engineering Journal 1 (2007) 27–42.

[22] T. Lengauer, R. E. Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Transactions on Programming Language Systems 1 (1) (1979) 121–141. doi:http://doi.acm.org/10.1145/357062.357071.

[23] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley Longman Publishing, Boston, MA, USA, 2006.

[24] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: Proceedings of the international symposium

on Code generation and optimization: feedback-directed and runtime optimization, CGO '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 75–.
URL http://dl.acm.org/citation.cfm?id=977395.977673

[25] C. Team, clang: a c language family frontend for llvm, http://clang.llvm.org/ (March 2012 (last accessed)).

[26] M. Hind, Pointer analysis: haven't we solved this problem yet?, in: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '01, ACM, New York, NY, USA, 2001, pp. 54–61.

[27] B.-C. Cheng, W.-M. W. Hwu, Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation, in: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI '00, ACM, New York, NY, USA, 2000, pp. 57–69.

[28] GNU, The internals of the gnu compilers, http://gcc.gnu.org/onlinedocs/gccint/ (2011).

[29] A. Milanova, A. Rountev, B. G. Ryder, Precise call graphs for c programs with function pointers, Automated Software Engg. 11 (1) (2004) 7–26. doi:http://dx.doi.org/10.1023/B:AUSE.0000008666.56394.a1.

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, IEEE 4th Annual Workshop on Workload Characterization.

[31] Standard performance evaluation corporation (spec), http://www.spec.org/benchmarks.html (2006).

[32] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, in: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 57–76. doi:http://doi.acm.org/10.1145/1297027.1297033.
URL http://doi.acm.org/10.1145/1297027.1297033