A Journal of Software Engineering and Applications, ***2013***, **,**-***
doi:10.4236/jsea.2013.***** Published Online *** 2013 (http://www.scirp.org/journal/jsea)

Source-to-Source Refactoring and Elimination of Global Variables in C Programs

Hemaiyer Sankaranarayanan, Prasad A. Kulkarni

Electrical Engineering and Computer Science, University of Kansas, Lawrence, Kansas, USA
Email: prasadk@ku.edu
Received February 15, 2013.

Abstract

A global variable in C/C++ is one that is declared outside a function, and whose scope extends the lifetime of the entire program. Global variables cause problems for program dependability, maintainability, extensibility, verification, and thread-safety. However, global variables can also make coding more convenient and improve program performance. We have found the use of global variables to remain unabated and extensive in real-world software. In this paper we present a source-to-source refactoring tool to automatically detect and localize global variables in a program. We implement a compiler based transformation to find the best location to redefine each global variable as a local. For each global, our algorithm initializes the corresponding new local variable, passes it as an argument to necessary functions, and updates the source lines that used the original global to now instead use the corresponding local or argument. In this work we also characterize the use of global variables in common benchmark programs. We study the effect of our transformation on static program properties, such as the change in the number of function arguments and program state visibility. Additionally, we quantify dynamic program characteristics, including memory and runtime performance, before and after our localizing transformation.

Keywords: Global variable; Program refactoring; Compiler transformations

1 Introduction

A *Global* variable is an external variable in C and C++ 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 determine 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 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 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 is 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 in the data region of the process address space, providing only one copy of these variables for all program threads. Fifth, global variables typically violate the *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. Sixth, global variables can create mutual dependencies and untracked interactions between different program components causing an irregularity, called *action at a distance* in software engineering. This issue arises when operations in one part of the program affect behavior in some other program part. 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: (a) Efficiency – 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. (b) Convenience – 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 of dislocated use of program variables, it may be difficult for the programmer to determine the best place for declaring the *local* variable and find the best path to make it available to all functions setting or using it. Such use of globals is especially attractive for developers updating unfamiliar code regions in large programs. However, given the harmful effects of global variables, it will be more desirable if we could provide developers the convenience of using global variables, but automatically *localize* them to preserve the dependability, understandability, verifiability, and maintainability of the source code program.

In this work, we develop and implement a compiler-based algorithm to automatically find and eliminate global variables in a program by transforming them into local variables. Our algorithm automatically finds the closest *dominator* function to localize each global variable, and then passes the corresponding local variable as a parameter to every function using the original global. Function prototypes are appropriately modified throughout the program to 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 function argument. In this paper, we also design several experiments to measure the effect of this transformation on the space and time requirements of the modified programs. Thus, we make the following contributions in this work:

1. To our knowledge, we construct 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 benchmarks.
3. We measure and quantify the effect of this transformation on the number of function arguments passed, along with its space and performance (time) overheads.

2 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] and individual programming practitioners [14] have derided and discouraged the use of global variables. At the same time, acknowledging the necessity and/or convenience of employing global variables/state, language designers have developed alternative programming constructs to 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 files [13]. Another construct that programmers often use in place of 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 maintenance that are generally associated with global variables [16].

To our knowledge, there exist only a few related attempts to automatically detect and eliminate global variables in high-level programs. Sward and Chamillard developed a tool to identify global variables and add them as *locals* to the parameter list of functions in Ada programs [17]. However, apart from operating only on Ada programs, this work does not describe their implementation and does not provide any static or runtime results. Yang et al. proposed and implemented a "*lifting*" transformation to move global variables into `main`'s local scope [18]. However, *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 embedded systems applications, and did not have to deal with most of the issues encountered in a more general technique to eliminate global variables.

More 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 Fortran MPI (Message-Passing Interface) programs [11]. Their transformation moves all globals into a single structure. Every MPI process gets its own instance of this structure, which is then passed as an argument to all functions. Thus, unlike our implementation, their transformation does not target or affect code maintainability. Additionally, this previous work also did not collect statistics on the use of global variables and the effect of the transformation on code maintainability and performance metrics. Smith and Kulkarni implemented a similar algorithm to transform global variables into locals to make 'C' programs thread-safe [10]. However, this work was not targeted at code maintainability and did not implement a source to source transformation. Moreover, it did not collect and analyze any statistics about global variables and the transformation as done in this paper.

The ultimate goal of our research is to develop a new code refactoring tool that can flexibly 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.

3 Localizing Global Variables

Even moderate-sized programs in C/C++ often 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. This tool provides command-line options that enables the user to selectively eliminate all or some particular global variables and instantly see the changes made to the source code files. In this section we provide more details on our compiler-based framework and transformation algorithm.

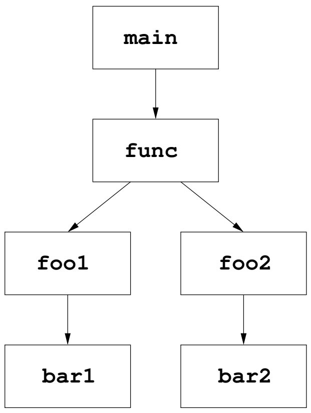
3.1 Transformation Algorithm for Localizing Global Variables

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 of the appropriate function, pass these new local variables to other functions using the original globals, update function headers and the variable names in the source statements accessing each global variable to instead use the new local/argument. We use the small example program in Figure 1(a) to explain our transformation algorithm in more detail. The syntax "`= var`" in Figure 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 1(b) shows the call-graph that will be generated for the example program in Figure 1(a).
2. The compiler then detects all global variables in the program, as well as 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 declared in application programs, 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 a particular 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 1(a) has the function `func()` as its common dominator.
4. Simply localizing each global variable in its closest common dominator function may compromise the semantics of the original program, if this dominator function is called multiple times. In such cases, the corresponding localized variable will be re-declared and re-initialized each time the dominator function is invoked, which is different than the single initialization semantics of the original global variable. For instance, in the example program in Figure 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`. Consequently, for each global variable, we traverse the dominator tree upwards starting from its closest common dominator 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 1(a), the global variable `var` is moved to the function `main()` and initialized as the new local variable `gbl_var`.
6. The next step involves finding all the functions in the call-graph between the legal dominator and the functions

```
int var;
int main(){
   func();
   func();
}
void func(){
   foo1();
   foo2();
}
void foo1(){
   bar1();
}
void bar1(){
    = var;
   var = ;
}
void foo2(){
   bar2();
}
void bar2(){
    = var;
   var = ;
}
```

(a) Original program using global variable



(b) Function call–graph

```
int main(){
   int gbl_var = 0;
   func(&gbl_var);
   func(&gbl_var);
}
void func(int *gbl_var){
   foo1(gbl_var);
   foo2(gbl_var);
}
void foo1(int *gbl_var){
   bar1(gbl_var);
}
void bar1(int *gbl_var}{
    = *gbl_var;
   *gbl_var = ;
}
void foo2(int *gbl_var){
   bar2(gbl_var);
}
void bar2(int *gbl_var){
    = *gbl_var;
   *gbl_var = ;
}
```

(c) Program after applying our refactoring transformation to localize global variables

Figure 1: Example to illustrate the program transformation to localize global variables

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 1(a), the local variable `glob_var` is passed by reference to all its frontier functions, namely `func()`, `foo1()` and `foo2()`.

7. Our transformation then modifies 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 interfaces of functions `bar1()` and `bar2()` are updated to accept the address of the local variable `gbl_var` as an argument.
8. Finally, our tool automatically updates every use of each global variable in the program statements to instead use its corresponding local variants. Thus, we can see all sets/uses of the global variable `var` replaced by the function argument `gbl_var` in the function bodies of `bar1()` and `bar2()`.

Thus, our algorithm to eliminate global variables automatically transforms the program in Figure 1(a) to the program in Figure 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.

4 Compiler and Benchmark 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 describe our compiler framework, existing framework limitations, and present the set of benchmark programs.

4.1 Compiler Framework and Limitations

We use the modern and popular Clang/LLVM [24, 25] compiler for this work. 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.

Clang/LLVM is a highly popular and heavily adopted compiler framework. However, the Clang frontend is still maturing and some less common compiler features/algorithms are not yet implemented in this framework. Such deficiencies impose a few restrictions on our current implementation.

The first limitation is on account of Clang's failure to generate precise call-graphs in the presence of function pointers. In our current work we circumvent this problem by supplementing the compiler-generated static call-graph with runtime profiling-based information to map indirect function call-sites with their targets for each benchmark-input pair. Our frame-

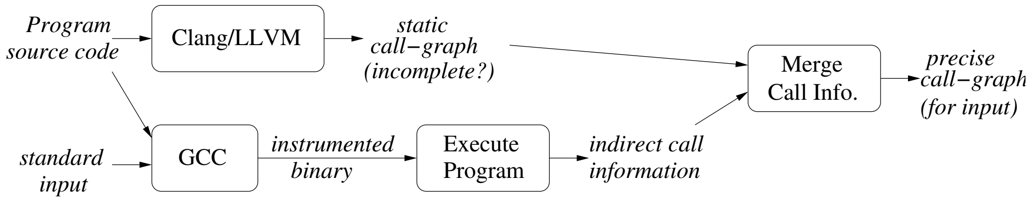


Figure 2: Framework for obtaining precise call-graph information for our analysis experiments

work for call-graph generation is illustrated in Figure 2. We modified GCC (version 4.5.2) [26] to instrument each source file with additional instructions that output the (*caller* → *callee*) function relationships at every indirect call on program execution. This supplemental indirect function call information is used to complete the static call-graph, when 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 [27].

Another related limitation is Clang's failure to correctly deal with variable *aliasing* in all cases. Aliasing occurs when a data location in memory can be accessed through different symbolic names. Our transformation tool is able to detect simple aliasing cases when a global variable is passed as a parameter to another function. However, we do not yet handle more complex aliasing cases in the source codes.

We emphasize that none of these shortcomings are fundamental restrictions on the algorithm, and will be resolved by providing better compiler support. Extending Clang to provide such support is part of our future work.

Benchmark	Total	RO/WO	Unused	RW	Moved
MiBench benchmarks					
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
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
SPEC CINT benchmarks					
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 1: Static number and type of global variables

4.2 Benchmark Suite

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 [28] and five benchmarks from SPEC CPU CINT2006 benchmark suite [29]. 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. The following MiBench benchmarks were analyzed but not included in our experimental set since they do not contain any *read/write* global variables: *basicmath*, *crc32*, *fft*, *patricia*, *qsort*, *rijndael*, *sha*, and *susan*. Additionally, *rsynth* from MiBench was not included as it produces no traceable output to verify the correctness of our transformation.

Table 1 shows the static characteristics of global variables in our selected benchmark programs. 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 unused 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 1 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 when using the corresponding function parameter pointers that are passed via reference. (b) Global variables used in functions called indirectly: We do not yet update function pointer declarations. (c) Miscella-

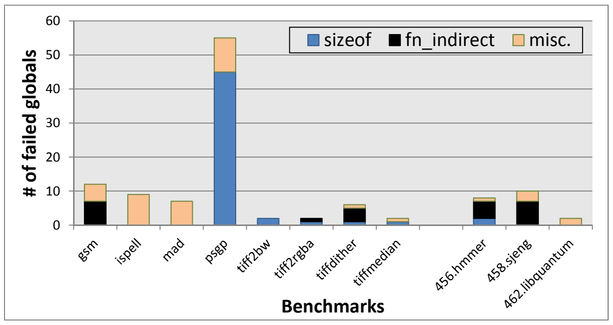


Figure 3: Categories and number of global variables that our tool fails to localize for our benchmark programs

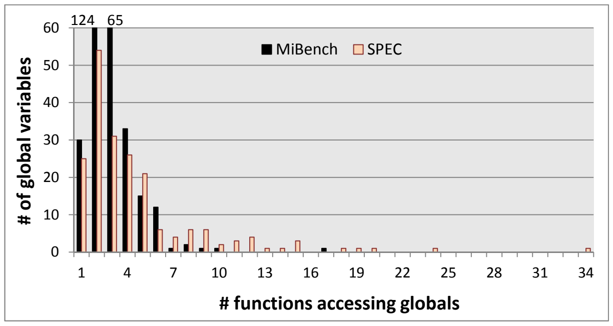


Figure 4: Number of functions accessing global variables

neous: Global variables that cause the compiler to generate incorrect code, if transformed. We found that most of the failed cases in the "miscellaneous" category occur due to the imprecise alias analysis performed by the Clang compiler. Please note that global variables belonging to the first two sets are automatically detected and bypassed by our tool, with a message sent to the user. We expect that the future implementation of a more precise alias analysis algorithm in the underlying compiler will enable our tool to automatically detect and correctly handle global variables in the third category. Figure 3 plots the number of failed *RW* global variables in each of these three categories for benchmarks that contain at least one failed *RW* global variable.

4.3 Properties of Global Variables

One typical use of global variables is as a convenience feature when particular program state is set or accessed in multiple program locations, and it is difficult to determine the best place to declare the variable and pass it as an argument so that it is visible in all program regions that need it. Global variables are also sometimes used to improve program efficiency by reducing the overhead of passing the variable to several different program functions. Figure 4 plots the number of functions that use/set each global variable. For example, the first set of bars in Figure 4 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. While this usage pattern is quite counter-intuitive, 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.

5 Experimental Results

In this section we describe and quantify the static and dynamic properties of our transformation to eliminate global variables. Our experiments employ the set of standard benchmark programs described in Section 4 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.

5.1 Static Characteristics of Our Transformation Algorithm

Our transformation to eliminate global variables can affect many static aspects of the high-level program. 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 3 to localize all the global variables in the *Moved* column of Table 1.

5.1.1 Effect on Average and Maximum Function Arguments

After localizing the global variables, our algorithm needs to make their state 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 5 and 6 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 average and maximum number of function arguments is not significantly affected for most of the benchmark programs, although this number increases substantially for a few 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

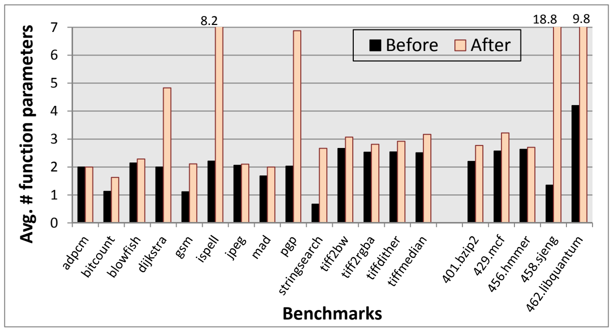


Figure 5: Average number of function parameters before and after applying our localizing transformation to eliminate global variables

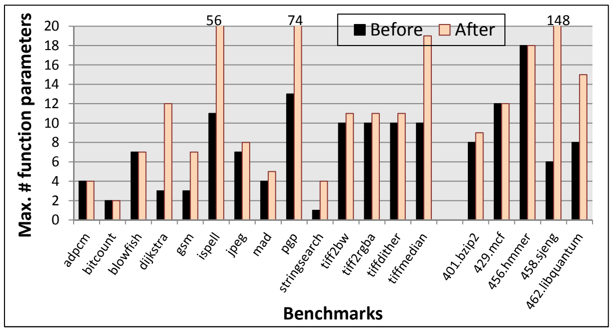


Figure 6: Maximum number of function parameters before and after applying our localizing transformation to eliminate global variables

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 that it makes the declarations of all variables used/set in any function explicit in each function header. This property is particularly important both from the aspects of program maintainability and verifiability. Unfortunately, passing additional function arguments can have an adverse effect on program efficiency. We explore the dynamic performance properties of our transformation in Section 5.2.

5.1.2 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 (functions `foo1()` and `foo2()` in Figure 1). Figure 7 presents the number of frontier functions for every transformed variable. The first set of bars in Figure 7 reveal that 12 of the new local variables in the MiBench benchmarks and no new local variable in the SPEC benchmarks have zero

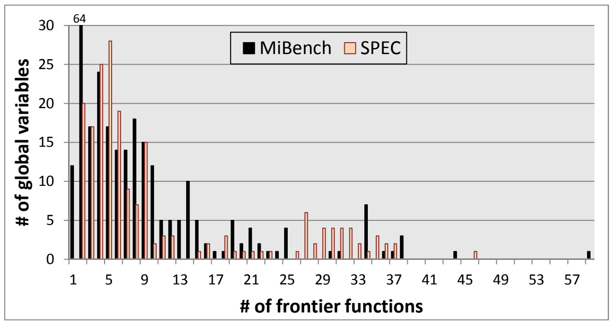


Figure 7: Number of frontier functions needed for the transformed local variables

frontier functions. Thus, we can see that most local variables have only a small number of frontier functions. This observations shows that many global variables are used in functions that are located close to each other in the static program call-graph. However, some globals 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 in the MiBench benchmarks that has 58 frontier functions and another in SPEC with 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 to use global variables. By automatically handling such scenarios, our tool allows the programmer the convenience of using global variables in difficult situations, but eliminates them later to satisfy software engineering goals.

5.1.3 Effect on Program State Visibility

Global variables are visible and accessible to all functions in the program. It is often argued that such global visibility makes it more difficult for automatic program verification and maintainability. One goal of our localizing transformation is to reduce the visibility of all variables to only the program regions where they are needed to assist verification and maintainability tasks. Figure 8 plots the percentage visibility of each transformed local variable as a ratio of the number of functions where the corresponding program state is visible to the total number of functions in the program. There is a point-plot for each transformed variable in Figure 8, sorted by its percentage visibility over all MiBench and SPEC benchmarks. Note that global variables are visible throughout (100%) the program. Thus, this figure shows that, after transformation, visibility is drastically reduced for most program state that was originally held in global variables. For example over 81% of the (original) 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.

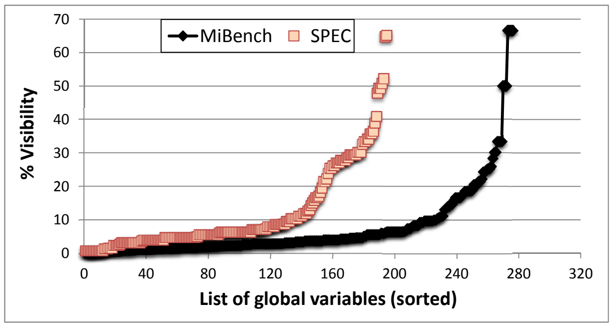


Figure 8: Percentage reduction in visibility of transformed variables compared to globals visible throughout the program

5.2 Dynamic Characteristics of Our Transformation Algorithm

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.
- 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 of the function activation records (stack).
- Global variables are 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 these experiments all our benchmark programs were compiled with GCC (version 4.5.2) [26] using the '-O2' optimization flag for the x86 32-bit platform running the Linux operating system. [1] 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]The original (before transformation) *tiff2rgba* benchmark program failed to run correctly with GCC's -O2 optimizations. Therefore, this program was run unoptimized with -O0 flag.

5.2.1 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, we can only count the dynamic instructions executed 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 effects.

5.2.2 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 to reduce the *data* space and expanding the 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 9 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 time, note that while a program only maintains one copy of any global variable, multiple copies of the corresponding local variable/argument may reside simultaneously on the stack for the transformed program. Therefore, there also exist programs, such as *adpcm*, *bitcount*, and *blowfish*, that show no discernible reduction in *data* size, but still encounter significant increases in maximum stack space use.

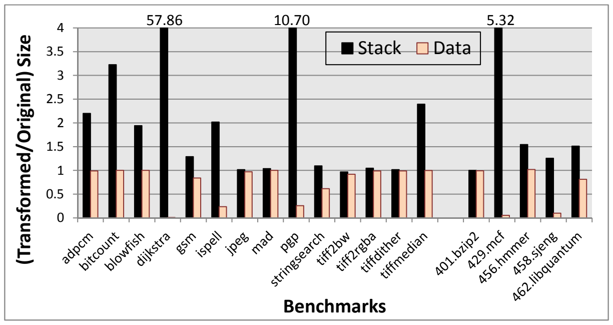


Figure 9: Ratio of the total *data* and maximum *stack* area consumed by each process at runtime before and after the localizing transformation

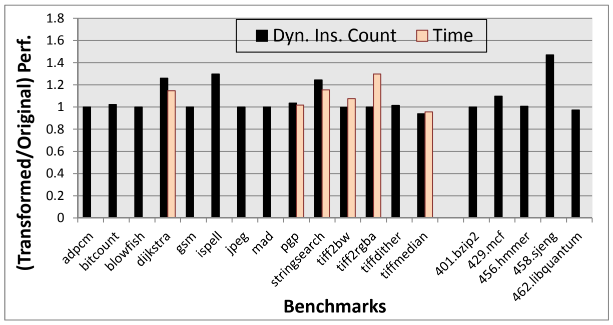


Figure 10: Ratio of *dynamic instruction counts* and program *run-time* before and after the localizing transformation

5.2.3 Effect on Dynamic Performance

Is this section we present experimental results 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 10 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 difference* (with a 95% confidence interval) [30] with and without our transformation. 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 typically 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 5). Several benchmark programs including *dijkstra*, *ispell*, *stringsearch*, and *458.sjeng* fall into this category. Interestingly, we observe that, in most cases, the increases in dynamic instruction count *do not produce* a corresponding increase in the actual benchmark runtime. The most notable exception to this observation is *tiff2rgba* that degrades substantially over the original program run-time after our localizing transformation. Remember that *tiff2rgba* is the only program not compiled with GCC's -O2 optimizations. Thus, it seems that optimizations performed by GCC and the x86 micro-architecture do a good job of reducing the overhead caused by our transformation to localize and eliminate global variables.

6 Future Work

There are a number of improvements that we plan to pursue in the future. First, we plan to implement more precise pointer analysis and improve alias analysis in the Clang/LLVM compiler, to appropriately resolve indirect function calls and build a precise call-graph for each benchmark. Second, this work only evaluates the case of localizing *all* global variables in a program, which can result in a large number of frontier functions and additional arguments. We are currently developing an Eclipse-based interactive framework to enable the user to selectively localize the most important global variables. It may also be possible to develop machine-learning algorithms to automatically find the most promising globals to localize based on certain user policies. Third, we will further investigate the causes of performance overhead and develop optimizations to reduce the overhead of the localizing transformation. 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. Finally, we require good metrics to evaluate the benefit of our tool during program development, dependence, maintenance, verification, and thread-safety. We plan to develop such metrics in the future.

7 Conclusions

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 the original global variable to now instead use the corresponding local or function argument. Our compiler based transformation tool is implemented using the popular Clang/LLVM compiler framework. We also analyze the static and runtime effects of our localizing transformation to allow the developer to make an informed decision regarding whether to localize any/all global variables. We found that many 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.

References

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

[2] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8:28–34, February 1973.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

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

[5] David Binkley, Mark Harman, Youssef Hassoun, Syed Islam, and Zheng Li. Assessing the impact of global variables on program dependence and dependence clusters. *J. Syst. Softw.*, 83:96–107, January 2010.

[6] Françoise 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, pages 145–154, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 50–59, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Sue Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13:263–279, September 2001.

[9] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Softw. Eng.*, 29:495–509, June 2003.

[10] Adam R. Smith and Prasad 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, pages 205–214, New York, NY, USA, 2011. ACM.

[11] Gengbin Zheng, Stas Negara, Celso L. Mendes, Laxmikant V. Kale, and Eduardo R. Rodrigues. Automatic handling of global variables for multi-threaded mpi programs. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 220–227, December 2011.

[12] Peter Sestoft. Replacing function parameters by global variables. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 39–53, New York, NY, USA, 1989. ACM.

[13] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

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

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

[17] Ricky E. Sward and 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, pages 29–34, New York, NY, USA, 2004. ACM.

[18] Xuejun Yang, Nathan Cooprider, and John 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, pages 60–69, New York, NY, USA, 2009. ACM.

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

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

[21] Dirk Wilking, Umar Farooq Khan, and Stefan Kowalewski. An empirical evaluation of refactoring. *e-Informatica Software Engineering Journal*, 1:27–42, 2007.

[22] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Language Systems*, 1(1):121–141, 1979.

[23] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2006.

[24] Chris Lattner and Vikram 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, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

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

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

[27] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.

[28] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

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

[30] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.