Localizing Globals and Statics to Make C Programs Thread-Safe

Adam R. Smith[*] Prasad A. Kulkarni
University of Kansas
Department of Electrical Engineering and Computer Science
Lawrence, Kansas, USA
smith195@illinois.edu; prasadk@ku.edu

ABSTRACT

Challenges emerging from the exponential growth in CPU power dissipation and chip hot spots with increasing clock frequencies have forced manufacturers to employ multicore processors as the ubiquitous platform in all computing domains. Embedded mobile devices are increasingly adopting multicore processors to improve program performance and responsiveness at low power levels. However, harnessing these performance and power benefits requires the construction of parallel programs, a task significantly more complex than writing sequential code. Parallel code development is also made more difficult by differences in the use of several programming language constructs. Therefore, it is critical to provide programmers with tools to ease the formidable task of parallelizing existing sequential code or developing new parallel code for multicore processors.

In this work we focus on the use of *static* and *global* variables that are commonly employed in C/C++ programs, the languages of choice for developing embedded systems applications. Unprotected use of such variables produces functions that are not thread-safe, thereby preventing the program from being parallelized. Manually eliminating global and static variables from existing sequential code is tedious, time-consuming and highly error-prone. While no good solution to this problem currently exists, researchers have proposed partial mitigation techniques that require massive changes to linkers and runtime systems. In this work we study the characteristics and effects of static and global variables in traditional benchmark programs, and propose, construct, and explore a compiler-based, semi-automatic and interactive technique to handle such variables and generate thread-safe code for parallel programs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

[*]Currently, a graduate student in the department of Computer Science at the University of Illinois, Urbana, IL

General Terms

Performance, Languages

Keywords

Thread-safe, Globals

1. INTRODUCTION

A microprocessor operating at a higher clock frequency consumes more power. The *power wall*, which is a limit to the amount of power that a microprocessor chip can dissipate without failing, proved to be an impregnable barrier to the further scaling of microprocessor clock frequencies, effectively ending the era of high-performance uniprocessor chips in the server and desktop PC domains. Chip manufactures now exploit the continuous growth in chip transistor count enabled by Moore's law by placing multiple simpler reduced clock-rate cores on a single processor die [2]. Advanced embedded applications are rapidly becoming more complex, and high-end embedded processors are facing the same power constraints that were earlier seen in PC processors. Consequently, several embedded processor manufacturers and application developers are advocating the use of symmetric multiprocessing (SMP) systems to realize the scalable performance and power requirements of existing and future embedded applications [11, 34].

Unfortunately, scalable performance on multicore processors can only be achieved by parallel applications distributing their tasks across multiple *threads*. Consequently, the migration to multicore processors requires software developers to address two formidable tasks: (a) generate new parallel applications, and (b) convert existing sequential programs to use multiple threads. Unfortunately, it has been observed that writing scalable and error-free parallel code is often substantially more difficult, time-consuming, and costly as compared to writing an equivalent serial program [20]. Furthermore, programming language constructs that are routinely employed during sequential coding may not be available or may require special handling during the generation of parallel code. Therefore, software tools that can automatically manage such common programming obstacles can greatly ease the task of producing multi-threaded parallel applications for software developers.

In this work we focus on the use of *static* and *global* variables in imperative language programs like C/C++. Global variables provide a convenient (albeit, occasionally error-prone) mechanism to share information between multiple functions in the same program. Static variables are also typically used to hold global state, but the compiler restricts their visibility to only the function or file where they are defined. Static and global variables are an indispensable programming construct employed by C/C++ developers in programs of all sizes. Indeed, the lack of these constructs will make it substantially more cumbersome and tedious to program in such languages.

By virtue of residing in the *data* region of the process address space, only a single copy of each global and static variable is shared among all threads in a multi-threaded process. Consequently, depending on their purpose and use, each static or global variable in a multi-threaded program needs to be assigned separate thread-local storage, or its use needs to be guarded (typically by using semaphores or mutexes) to enforce synchronized and atomic access by each thread. However, manually eliminating or synchronizing the use of statics and globals is extremely hard, time-consuming and tedious. At the same time, several programming idioms can be naturally and more efficiently expressed using static and global variables. Therefore, researchers are exploring various mechanisms to appropriately manage static and global variables in multi-threaded programs without substantially restricting their use or significantly changing their semantics. Most of these mechanisms require major changes to existing threading libraries and the runtime system to provide a new and *distinct* global storage area for each thread in the process address space [33, 14, 13]. Additionally, in most cases the user still needs to use special keywords to help the compiler/runtime place static and global variables in their appropriate storage locations.

In this paper we explore a new compiler-based semi-automatic and interactive transformation to sanitize the use of static and global variables in multi-threaded C applications. Our approach is semi-automatic since, similar to existing techniques, the user is required to indicate the *category* of each static or global variable as *thread-local*, *shared*, or *other*. These categories are explained in more detail in section 4. We develop an interactive framework to simplify this categorization process. To handle variables in the *thread-local* category, our compiler-based transformation constructs the program call-graph, finds the definitions and uses of each variable, and determines the closest *dominator* function to create a thread-local copy for each such variable. The tool then moves the static/global variable as a local into this dominator function, correctly initializes its stack storage, and then passes this variable by reference to reach each function where it is needed. *Shared* category variables can be locally handled by synchronizing their accesses using semaphores. Variables in the *other* category require manual support. In this paper we describe the procedure, and explore the tradeoffs and limitations of handling the more challenging *thread-local* category of static/global variables. In contrast to existing approaches for managing thread-local data, our method requires no modifications to the language specifications, linker, loader, libraries, or the runtime system. Thus, a successful and low overhead implementation of the proposed approach will not only ease the immense task of converting existing serial C/C++ applications to their parallel versions, but also enable the continued use of static and global variables to simplify the writing of new parallel programs without additional language or runtime support. In this paper we investigate the challenges in achieving this goal. Thus, we make the following contributions in this paper:

1. Present characteristics on the use of static/global variables in popular embedded and high-performance benchmarks,
2. Describe a novel semi-automatic procedure to transform variables with global process scope into variables with thread-local scope to enable correct program parallelization,
3. Explore the space and performance trade-offs, as well as identify the challenges for precise application of this transformation, and
4. Illustrate the use of the viewer to enable user communication with the compiler transformation.

2. RELATED WORK

Increasing availability of multicore/multiprocessor machines and the growing demand for multi-threaded programs has resulted in several approaches to ease the task of writing parallel applications. In this section we describe previous efforts for providing thread-local storage, specifications for parallel languages, and other library, tool and runtime level support for parallel programming.

Several high-level languages provide standard or non-standard support for thread-local variants of traditional global and static variables. For example, Delphi provides the variable specifier `threadvar` [27], GNU C/C++ provides the `__thread` specifier [33], the unofficial new standard for C++ (C++0x) provides the `thread_local` specifier [14], and C# allows variables to be marked with the *ThreadStatic* attribute [13] to specify thread-local storage for global data. In contrast, our interactive approach does not require any changes to the language specification. Additionally, more than the language design, in this paper we focus on the implementation aspects of providing thread-local storage for static and global variables.

Library and compiler implementation support is also available to create thread-specific data. Thus, the POSIX thread interface [10], Windows libraries [4], and the boost C++ libraries [3] provide API functionality to create and manage thread-local storage for statics and globals. Apart from learning a new API, such implementation approaches are often cumbersome to use [14] and require significant support from the linker, system libraries and the runtime system [33]. Our tool employs a complementary and novel compiler-based approach that not only provides thread-local storage without any modifications to the library, linker, or runtime support, but can also automatically synchronize access for shared global variables.

Researchers have also developed new programming languages and extensions to existing languages to enable easier specification of parallelism [15, 30, 23, 31, 12, 36], Parallel programming languages typically present a significantly different programming interface than their sequential counterparts, and mark a steeper learning curve for many programmers. Traditional low-level multithreading libraries, such as *pthreads*, allow the programmer to express parallelism, but leave program and data partitioning, synchronization, communication and deadlock management in the hands of the programmer [25]. Concurrent programming has also been long practiced in the scientific computing domain using data parallel language extensions and message passing libraries [17, 16]. Our work is an attempt to sanitize global/static variables while maintaining the same programming interface.

Automatic compiler-based program parallelization achieved considerable success on regular scientific programs by extracting DOALL parallelism [21] from loop nests accessing arrays [7, 35]. Most of these techniques employ intraprocedural analysis to exploit fine-grained and vector parallelism, but were often not able to achieve encouraging results for general-purpose applications. Inter-procedural analysis required for coarse-grained parallelism is generally considered more complicated, and has only achieved limited success [18]. We are not aware of any automated inter-procedural technique that attempts to globally modify function declarations, as is done by our technique.

Bridges et al. propose a new source annotation and compiler directive, called *Commutative*, to enable concurrent invocation of functions with the property that multiple calls to that function are interchangeable even though they share internal state [8]. Rinard et al. developed a novel commutativity analysis to automatically detect operations or functions that generate the same final result regardless of the order in which they execute [32]. Although much different than our techniques, such analysis can be employed in our framework to support the handling of the *shared* category of static and global variables that occur in commutative functions.

Some of our techniques are also related to the popular compiler optimizations of scalar expansion and scalar privatization [21]. Scalar expansion replaces a loop scalar with a compiler generated temporary array that has a separate location for each loop iteration. Loop privatization is a slightly different way to achieve the same result by declaring the local as *private* to each iteration of a loop. Application of these techniques allow the loop to be vectorized or parallelized at the cost of increased code size. However, as opposed to our technique outlined in this paper, scalar expansion and privatization achieve different goals, are transformations local to a function and do not affect the calling interface of any function.

The ultimate goal of this effort 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 [28] and extensibility [22]. None of these tools provide an ability as yet to transform global/local variables, as we propose in this work.

3. IMPLEMENTATION AND USE OF STATIC / GLOBAL VARIABLES

In this section, we present a brief primer on the purpose and implementation of static and global variables in imperative languages like C and C++.

3.1 Statics/Globals in the Process Address Space

A *global* variable declaration is one that exists outside the scope and outlasts the life of any function. Global variables generally enjoy visibility throughout the program. The *static* specifier can be used during the declaration of any variable to extend its life for the entire program execution. In spite of their global lifetime, static variables enjoy restricted visibility that depends on where that static has been declared. A static variable declared outside a function definition has its scope restricted to the file where it is declared. In contrast, static variables declared in a function are only visible within that single function. All static and global variables are only initialized once and retain their value across calls allowing the program to store global state, whenever required.

Due to their global lifetimes, the compiler detects all static and global variables during program compilation and allocates them space in the data area of the process address space. Figure 1 shows a program snippet to illustrate the use of local (`loc_i`), function argument (`prm_i`), static (`stc_cnt`) and global (`gbl_iter`) variables. Depending on the variable kind, the compiler allocates them space in different regions of the process address space, as illustrated in Figure 2(a). While local variables and function arguments are allocated space in the function activation record on the process *stack*, global and static variables reside in the *data* region of the address space. Variables in the function activation records are reassigned space on every function invocation and reclaimed on function return. In contrast, only one instance of each static and global variable is available at any instance during program execution.

For a process with multiple threads, each thread gets its own stack space (along with a few other thread-level values maintained by the operating system), but shares the remaining regions with the other threads in the same process. Consequently, each thread gets its own copy of local variables and function arguments, and so these are referred to as *thread-local* storage. However, by virtue of residing in the *data* region of the process address space, a single

```
int gbl_iter = 10;     void foo(int prm_i) {
                          static stc_cnt;
void main() {
   int loc_i = 0;         if(prm_i < gbl_iter){
                             ....
   foo(loc_i);               stc_cnt++;
}                            foo(prm_i+1);
                          }
                       }
```

Figure 1: Declaration and Use of Static Variables

copy of each static and global variable is shared by all threads. The process address space for a multi-threaded process, and the sharing of variables in the *data* region is illustrated in Figure 2(b). Therefore, for correct program operation, updates to these static and global variables need to be synchronized between threads, or such variables should be converted into thread-local storage. In this paper we describe a novel method that employs an interactive compiler-based transformation to appropriately handle static and global variables in multi-threaded programs.

4. OUR APPROACH TO LOCALIZE GLOBAL AND STATIC VARIABLES

In this section we outline our general approach to privatize global and static variables to construct thread-safe programs. In section 4.1 we categorize global/static variables in single-threaded programs based on how they should be handled in thread-safe multi-threaded programs. Next, we employ an example program in section 4.2 to describe the actions to be taken by the user and the work that can be performed automatically by our tool to make programs thread-safe. Finally, in section 4.3 we present further details on the techniques and algorithms that we use during our transformation.

4.1 Categories of Statics and Globals

Depending on their use and program implementation, statics and globals in multi-threaded programs can be categorized as follows:

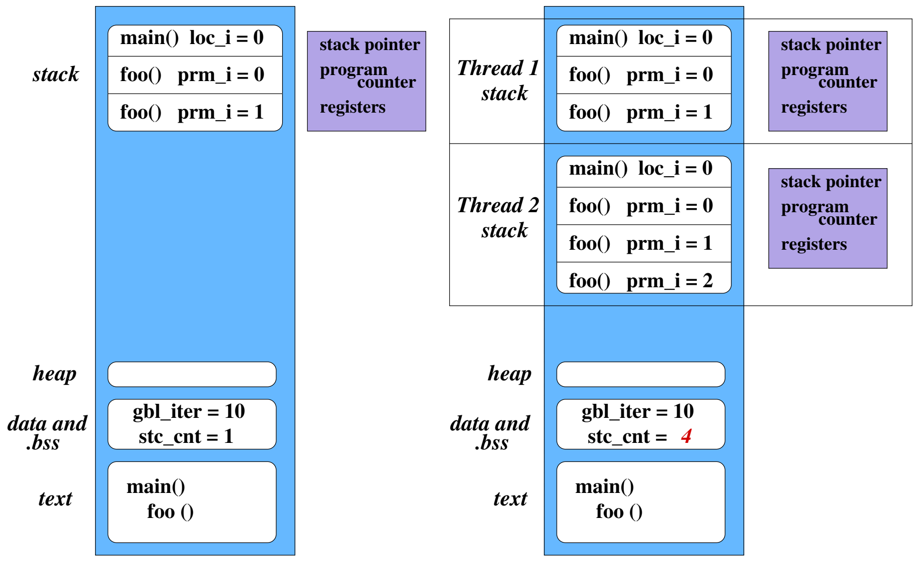
Category I: Global/static variable that holds and manipulates unrelated and independent values in each thread of the process. Our tool automatically assigns distinct thread-local storage to such variables.

Category II: Global/static variable that holds a value that is shared between threads, and the value is managed by the threads in no particular order. Our tool can automatically synchronize write access to such variables using a semaphore-like mechanism.

Category III: Global/static variable that holds a shared value, which must be managed by individual threads running in a specific order. This group of variables is difficult to handle automatically and may require manual user support, including changing the parallel algorithm.

Our compiler-based tool requires user input for correct program transformation. Therefore, we use the the VISTA interactive viewer [24] to query the category of each static and global variable from the user. Our tool can then automatically handle static/global variables in categories I and II to generate a thread-safe version of the program, but leaves variables in the final category unchanged.

4.2 Separation of Manual and Automatic Work During Our Transformation



(a) Single thread (b) Multiple threads

Figure 2: Process Address Space for Single and Multi-Threaded Programs

Our transformation is only targeted to make programs thread-safe and cannot automatically parallelize a sequential program. Therefore, in addition to specifying the category of each global and static, the user is also responsible for writing the parallel program that can create multiple threads and distribute the work amongst these threads. In this section we employ a hypothetical program to illustrate the distribution of work done manually by the user as compared to that performed automatically by our tool. We will also use this example to explain the workings of our transformation.

The example program presented in Figure 3(a) is derived from the main loop of a realistic sequential compiler. The compiler loop transforms the input code by reading, optimizing, and writing out functions one at a time. The original sequential program consists of all source lines in Figure 3(a), except those indicated in a bold font. The program uses three global variables: `in_fp` to hold the current location in the source file, `out_fp` to point to the next location in the output file for the compiled code, and `fn`, which points to the structure containing information regarding the current function to optimize. Based on user options, the `compile()` function can either send the function for low-quality fast transformation or high-performance optimization. The functions `opt1()` and `opt2()` use the global variable `fn` to optimize the code.

On multi-processor machines, the user can achieve faster compilation by concurrently transforming multiple functions from the input file on separate cores. In order to achieve this goal using our framework, the programmer is first required to manually port the sequential application to create and use multiple threads to compile different functions in parallel. For our example program in Figure 3(a), the user can create a parallel variant of the original program by adding the lines of code indicated by a bold font to this sequential program, and replacing the function call to `compile()` in function `main()` to instead call function `compile_thread()`. The new parallel program creates a new thread to compile each input function. Given multiple processing units, each parallel program thread still reads the input function and writes the transformed code sequentially, but can perform the optimization step in parallel. However, the parallel program in Figure 3(a) is not thread-safe due to its unprotected use of global variables. Therefore, its execution will most likely produce incorrect output code. In the following section we describe how our automatic tool can be employed to make this parallel program thread-safe.

4.3 Algorithm for Generating Thread-Safe Programs

Our tool accepts a non-thread-safe parallel program from the user. We have interfaced our compiler-based transformation with the interactive VISTA viewer to get two more pieces of information from the user. First, our tool requires the user to indicates the *entry* function for each (set of) thread (function `compile` in Figure 3(a)). The user can simply click on the entry function node in the call-graph structure presented by the viewer, and as displayed in Figure 4 for the program in Figure 3. In our current implementation, the user can only specify a single thread entry function at a time. Extending our tool to specify multiple functions for threads starting from different entry points should be straight-forward, and we leave its implementation as a part of future work. Second, our tool detects all global and static variables used in the subtree rooted at the thread entry function and presents this list to the user. The user is then expected to correctly specify the category of each static and global variable, as described in section 4.1. For this example, the variable `fn` belongs to category I, while variables `in_fp` and `out_fp` belong to category II.

Presented with the set of input source files, an indication of the thread entry function and a description of the category of each global/static variable used in the relevant part of the program, our tool can then proceed with the transformation to make the program thread-safe. Our transformation algorithm carries out different steps to handle each category of global and static variables for generating a thread-safe program. Variables belonging to category I require more extensive handling. The steps for managing category I globals and statics are described below:

1. The compiler scans the source files to generate a call-graph for the application. During the call-graph construction, the compiler stores information regarding the definitions and uses of all relevant global and static variables. At the same time,

```
FILE *in_fp;
FILE *out_fp;
struct FN *fn;

int main() {
   while (more_functions){
      compile();
      // compile_thread();
   }
}

void compile_thread() {
   // create new thread
   create_thread(&compile);
}

void compile() {
   read_next_function();
   if(user_opt == 1)
      fast_optimize();
   else
      optimize();
   write_opt_function();
}

void read_next_func() {
 // set fn by reading next
 // function from file in_fp
}

void write_opt_func() {
 // write optimized function
 // fn to file out_fp
}

void fast_optimize() {
   opt1();
}

void optimize() {
   opt1();
   opt2();
}

void opt1(){
 // optimize and modify function
 // pointed to by fn
}

void opt2(){
 // optimize and modify function
 // pointed to by fn
}
```

(a) Non-Thread-Safe Program

```
FILE *in_fp;
FILE *out_fp;
// struct FN *fn;

int main() {
   while (more_functions){
      compile_thread();
   }
}
void compile_thread() {
 // create new thread
   create_thread(&compile);
}

void compile() {
   struct FN *loc_fn = 0;
   read_next_function(&loc_fn);
   if(user_opt == 1)
      fast_optimize(&loc_fn);
   else
      optimize(&loc_fn);
   write_opt_function(&loc_fn);
}

void read_next_func(struct FN **p_fn) {
 // lock file in_fp
     // set p_fn by reading next
     // function from file in_fp
 // unlock file in_fp
}

void write_opt_func(struct FN **p_fn) {
 // lock file out_fp
     // write optimized function
     // p_fn to file out_fp
 // unlock file out_fp
}

void fast_optimize(struct FN **p_fn) {
   opt1(p_fn);
}

void optimize(struct FN **p_fn) {
   opt1(p_fn);
   opt2(p_fn);
}

void opt1(struct FN **p_fn){
 // optimize and modify function
 // pointed to by p_fn
}

void opt2(struct FN **p_fn){
 // optimize and modify function
 // pointed to by p_fn
}
```

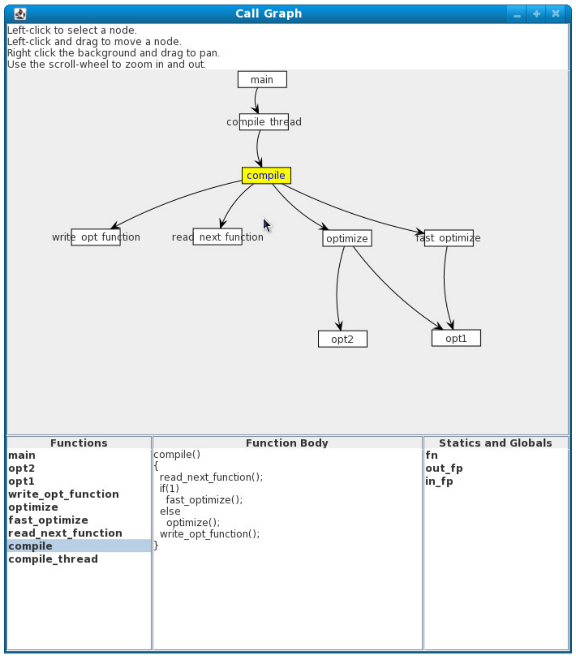
(b) Multi-Threaded Thread-Safe Program

Figure 3: Example of a sequential program with its semantically equivalent thread-safe multi-threaded version

the compiler also determines the data type and initialization values for each global and static variable.

2. For each global/static variable, our tool uses the Lengauer-Tarjan algorithm [26] to find the *closest dominator* function for the set of functions that use that particular variable. 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* [5]. For static variables, the dominator function is simply the immediate dominator of the function containing the static. The global dominators are more difficult to determine, since a global may be used in any number of functions. The global dominator is then determined by locating the first common dominator of all the functions that use that global. Thus, as an example the global variable `fn` that is used in two functions, `opt1()` and `opt2()`, in Figure 3(a) has the function `compile()` as its common dominator. The transformation will merge aliased locations into a single location for the purpose of finding dominators.

3. Next, our transformation moves each global/static variable as a *local* variable to its dominator function. Additional space is created on the stack of the dominator function for this definition of the local. The transformation also adds new in-



Left-click to select a node.
Left-click and drag to move a node.
Right click the background and drag to pan.
Use the scroll-wheel to zoom in and out.

Functions:

```
main
opt2
opt1
write_opt_function
optimize
fast_optimize
read_next_function
compile
compile_thread
```

Function Body:

```
compile()
{
 read_next_function();
 if(1)
   fast_optimize();
 else
   optimize();
 write_opt_function();
}
```

Statics and Globals:

```
fn
out_fp
in_fp
```

Figure 4: Viewer window showing the call-graph for the program in Figure 3(b)

structions to this function to correctly initialize the new local variable. In the program displayed in Figure 3(a), the global variable `fn` is moved to the function `compile()` and initialized to zero. Since the local variables reside on the stack, we can now ensure a distinct copy of each static and global variable for every thread.

4. The next step involves finding the functions in the call-graph between the common dominator and all the functions where the global/static is used. We call this set of functions as the global/static's *frontier*. The local copy for each global/static 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 3(a), the local copy of global `fn` is passed by reference to all its frontier functions, namely `fast_optimize()` and `optimize()`.

5. The next step in our transformation is to modify the calling interface of the end functions for each global/static to get the additional arguments corresponding to the thread-local variants of globals and statics. Thus, the calling interface of functions `opt1()` and `opt2()` is updated to accept the address of the local variant of `fn` as an argument. Finally, our tool also updates every use of each global/static to instead use the corresponding local/argument.

Global and static variables belonging to category II can be more easily handled by maintaining them as globals, but synchronizing their accesses using a semaphore-like mechanism. Thus, the application of our tool transforms the non-thread-safe parallel program in Figure 3(a) to the thread-safe parallel program in Figure 3(b). The parts of the program that are modified by our transformation are indicated in bold fonts.

5. OBSERVATIONS AND RESULTS

In this section we present our observations regarding the use of globals and statics in benchmark programs. We further describe the code changes and performance effects of our transformation.

5.1 Experimental Framework

The research described in this paper can be conducted as a source-to-source or a source-to-binary transformation. Our existing prototype uses the VISTA (VPO Interactive System for Tuning Applications) [24] interactive compilation framework. VISTA employs the EDG compiler frontend and the VPO (Very Portable Optimizer) compiler backend [6], and provides an intuitive interface for users to interact with the compiler backend. The backend VPO compiler performs its analyses and optimizations on a low-level program representation called Register Transfer Lists (RTL). Our configuration of VPO for this work generates code for the ARM architecture. The VISTA user interface can be used to view the program source or its assembly form, to select VPO optimizations and to see the state of the program at various stages of the compilation process. We have integrated our prototype implementation for handling statics and globals in multi-threaded programs within VPO, and have extended the VISTA user interface to enable the user to view the definition and uses of global and static variables and select their category as detailed in section 4.

Our selected benchmark set for this study include six benchmarks from the MiBench benchmark suite, which are C applications targeting specific areas of the embedded market [19]. We also include all C programs (except `462.libquantum`, which did not build with the default version of our compiler frontend) from the CPU-intensive SPEC CPU CINT2006 benchmark suite [1] to experiment with more complex applications. Table 1 contains descriptions of our benchmark programs.

5.2 Characteristics of Statics and Globals in Benchmark Programs

Global and static variables are very commonly employed in most C/C++ programs. In this section we study various statistics on the use of such variables in our benchmark programs, and their effect on the thread-safety of each function. We believe that such information is important to understand the factors to be considered during the construction of complex thread-safe parallel programs.

Table 2 reveals several properties of statics and globals in our benchmark programs. The first column lists the benchmark name. The next three columns present the number of read/write-only (RWO), read and write (RW), and unknown (UK) static variables. Read/write-only statics do not require any further handling since a single copy of such variables can be harmlessly shared between all threads. Read and write statics may need to be assigned separate storage for each thread. The unknown (UK) category includes those variables for which we are unable to derive any access information. Further analysis reveal that in most such cases only the address of the variable is used or is passed as an argument to some other function(s) that may manipulate its value. The next three columns present similar information regarding the number of read-only or write-only (RWO), read and write (RW) and unknown (UK) global variables. In addition, we observe that a significant number of global variables were declared but never used in any function *reachable* from `main()` in our call-graph.[1] This number is listed in the next column (labeled *US*). Thus, we can see that most of our benchmarks make generous use of static, and in particular, global variables. Moreover, the use of such variables becomes more prominent with grow-

[1]One important reason for unreachable functions, which is explained shortly, is the presence of indirect function calls.

Benchmark	Suite	LOC	Funcs.	Description
400.perlbench	SPEC CINT	43,722	1,984	cut-down version of the scripting language Perl v5.8.7
401.bzip2	SPEC CINT	2,852	103	popular compression program v1.0.3
403.gcc	SPEC CINT	145,410	5,675	based on gcc Version 3.2 for Opteron
429.mcf	SPEC CINT	699	24	network simple algorithm for vehicle scheduling
445.gobmk	SPEC CINT	39,106	2,681	plays the simple but complex AI game Go
456.hmmer	SPEC CINT	12,039	540	protein sequence analysis using hidden Markov models
458.sjeng	SPEC CINT	4,935	145	a highly-ranked chess program
464.h264ref	SPEC CINT	16,736	591	a reference implementation of H.264/AVC
adpcm	MiBench	114	4	compress 16-bit linear PCM samples to 4-bit samples
blowfish	MiBench	299	8	symmetric block cipher with variable length key
dijkstra	MiBench	70	7	Dijkstra's shortest path algorithm
ispell	MiBench	2,802	112	fast spelling checker
jpeg	MiBench	1,256	62	image compression and decompression
stringsearch	MiBench	134	10	searches for given words in phrases

Table 1: Our set of benchmark programs (LOC – counts the number of lines containing a semi-colon; Funcs. – counts the number of static function definitions in each program).

Benchmark	Statics			Globals				Reach	Non-thread-safe Fn.		
	RWO	RW	UK	RWO	RW	UK	US	Funcs	Statics	Globals	Total
400.perlbench	4	2	5	31	277	17	208	608	322	515	517
401.bzip2	0	0	0	8	8	0	14	81	0	16	16
403.gcc	19	49	4	187	919	34	386	3890	2552	3174	3178
429.mcf	0	0	0	0	6	1	1	24	0	6	6
445.gobmk	5	22	2	26	264	12	10111	716	201	554	558
456.hmmer	6	15	1	15	11	2	14	208	15	29	32
456.sjeng	17	5	0	42	161	4	26	120	3	101	101
464.h264ref	37	8	7	77	355	8	88	504	32	334	335
adpcm	0	0	0	2	3	0	0	3	0	2	2
blowfish	0	0	0	0	1	0	1	4	0	2	2
dijkstra	0	0	0	1	9	0	0	7	0	6	6
ispell	1	5	2	4	83	1	15	108	50	84	84
jpeg	0	1	0	0	3	1	1	19	3	5	5
stringsearch	0	0	0	0	3	0	5	3	0	3	3

Table 2: Statistics and effect of global and static variables

ing program size, which indicates the benefit of these variables for writing complex code. Our approach to make the program thread-safe converts static and global variables in the columns *RW* and *UK* by either synchronizing their accesses or by moving them to their respective dominator functions as locals, initializing them and passing them as arguments to all their end-point functions.

The next column, labeled *Reach Funcs*, shows the count of the number of functions in the call-graph for each benchmark that are reachable from the start function `main()`. The final three columns in Table 2 present the number of non-thread-safe functions (considering `main()` as the thread entry function) in each benchmark program due to the presence of *statics*, *globals* and their union (*total*). We consider a function as non-thread-safe if it uses a global or static variable in the *RW* or *UK* categories, or if it calls a non-thread-safe function. By this metric, a significant majority of reachable functions in most of our benchmark programs are non-thread-safe. This metric demonstrates the importance of appropriately addressing this issue to generate parallel programs for current and future shared-memory multi-processor machines.

Our approach to localize category I static and global variables requires the construction of a precise call-graph for each program. The presence of indirect function calls via function pointers makes precise call-graph construction difficult in languages like C/C++. Table 3 shows statistics regarding the use of indirect function calls in our benchmark programs. The columns labeled *Num* and *Funcs* respectively present the number of indirect function calls and the number of unique functions invoking such calls for each benchmark program. Thus, we find that most larger benchmarks make generous use of indirect function calls. Unfortunately, the VPO compiler does not yet perform the pointer analysis necessary for the proper resolution of function pointers. This limitation is a source of inaccuracy in some of our current results. However, please note that this aspect is not a limitation of the general technique, since precise function pointer analysis and call-graph construction has been shown to be feasible for most programs in earlier studies [29].

The next two columns in Figure 3 show the number of *Total* and *Orphan* functions in each benchmark program. *Total* functions is the number of nodes in the call-graph for each function. Our compiler creates a call-graph node for every distinct function defined or invoked in the benchmark program, and thus also includes library functions. *Orphan* functions are those that cannot be reached from the `main()` function and are, therefore, purged from the call-graph. The large number of *orphan* functions is partly an artifact of the limitation of our current call-graph construction algorithm, which is unable to resolve indirect function calls producing no edges for such calls in the call-graph. This limitation in our compiler framework will be resolved as part of future work by providing a more precise implementation of function pointer analysis.

Benchmark	Indirect Calls		Functions		Benchmark	Indirect Calls		Functions	
	Num	Funcs	Total	Orphan		Num	Funcs	Total	Orphan
400.perlbench	137	99	1982	1330	401.bzip2	20	5	121	24
403.gcc	461	239	5684	1729	429.mcf	0	0	39	1
445.gobmk	44	20	2726	1968	456.hmmer	8	4	601	342
456.sjeng	1	1	176	26	464.h264ref	368	29	636	88
adpcm	0	0	8	1	blowfish	0	0	16	4
dijkstra	0	0	15	0	ispell	0	0	171	4
jpeg	108	37	88	47	stringsearch	0	0	19	13

Table 3: *Number* of indirect function invocations and number of *Functions* containing indirect calls per benchmark

Benchmark	Statics		Globals		Stc Frontier		Gbl Frontier		Arguments	
	Num	Size	Num	Size	Max	Avg	Max	Avg	Before	After
400.perlbench	7	216	294	12484	501	352.14	494	360.49	1.77	181.64
401.bzip2	0	0	8	2133	0	0.00	9	2.75	2.15	2.62
403.gcc	53	503	953	612197	1517	287.23	2500	274.48	1.96	73.95
429.mcf	0	0	7	6148	0	0.00	0	0.00	2.50	2.88
445.gobmk	24	70919	276	1687299	174	47.50	425	78.97	2.46	35.95
456.hmmer	16	348	13	81651	3	1.06	8	1.62	2.63	3.03
456.sjeng	5	20	165	2572792	0	0.00	35	7.96	1.32	18.18
464.h264ref	15	1272	363	186394	11	3.33	192	8.32	1.90	10.32
adpcm	0	0	3	2504	0	0.00	0	0.00	1.33	2.33
blowfish	0	0	1	4168	0	0.00	0	0.00	3.75	4.00
dijkstra	0	0	9	40828	0	0.00	1	0.11	2.00	4.00
ispell	7	8585	83	32848	33	9.86	55	7.11	2.19	10.54
jpeg	1	4	4	188	0	0.00	0	0.00	1.84	2.32
stringsearch	0	0	3	1032	0	0.00	0	0.00	0.67	2.67

Table 4: Static results of the transformation to localize static and global variables

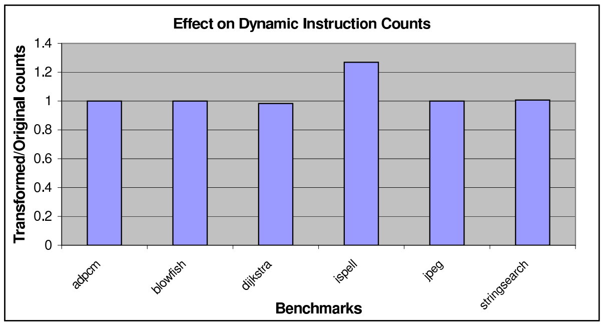
5.3 Static Code Changes and Dynamic Effects of Our Transformation

In this section we attempt to evaluate the impact of our approach in terms of static code changes and dynamic performance effects. However, measuring the effects of our transformation on program performance is non-trivial. Similar to all existing schemes to handle global/static variables in multi-threaded programs, application of our transformation also requires user knowledge to update the original sequential program to create and use threads, identify the thread entry functions, and categorize each global and static variable, as described in section 4. In the absence of suitable multi-threaded benchmarks and necessary program knowledge, we make the following assumptions for our results with the chosen single-threaded benchmarks in this section: (1) The function `main()` is selected as the entry point for each thread, which means that *all* read-write static and global variables in the entire program need to be handled based on their category. (2) All relevant static and global variables belong to category I, which requires all such variables to be converted into thread-local storage as explained in Section 4.3. We believe that the handling of category I globals provides the most interesting and novel component of our transformation. (3) We do not manually parallelize our sequential benchmark programs (as described in Section 4.2). Therefore, we only measure the static and runtime statistics of the single-threaded program before and after our transformation. The measured statistics thus quantify the scenario where *all* global and static variables are localized and all reachable program functions are made thread-safe.

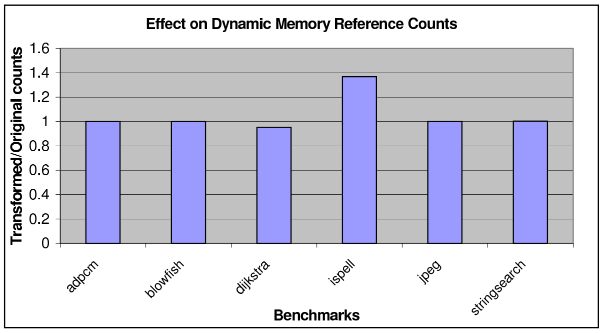
Table 4 presents the static results of applying our transformation to the benchmark programs. For each benchmark, columns two and three show the number and total size (in bytes) of static variables that are localized. Similarly, the next two columns show the number and size of global variables that are assigned thread-local storage. Thus, the columns labeled by *Size* indicate the additional number of bytes that need per-thread allocation on the stack (in their respective dominator functions), and have to be initialized by each thread at runtime. Our transformation adds instructions to allocate space and initialize this new stack storage. Such variables also need to be passed around as *additional* arguments from their point of declaration to the respective functions where they are set/used. Thus, precise inter-procedural and alias-based detection of global and static variable access types (read/write) is critical to accurately determine the number of variables to handle and limit the performance cost of this transformation.

Our transformation modifies each function interface to accept the additional number of function arguments. The next four columns in Table 4 list the *maximum* and *average* number of *frontier* functions for each benchmark. Frontier functions reside between the declaration and use points for each static/global and transfer the variables between these points. Our algorithm attempts to place each static/global variable in the function that is closest to its points of use to minimize the number of frontier functions and associated argument passing overhead. Even then, several (particularly, larger) benchmarks contain large number of frontier functions. The final two columns show the number of average function arguments before and after our transformation. Thus, the average number of function arguments is seen to increase dramatically in many cases and tracks the number of frontier functions.

Finally, we present results of experiments that quantify the performance effect of our transformation. Unlike static and global variables that are implicitly zeroed by the operating system, local variables need explicit initialization within the program. Thus, our transformation not only requires additional storage space on the process stack, but also adds instructions to initialize the new locals in the respective dominator functions. New instructions are also needed to pass arguments around. The number and kind of such instructions depend on the machine's calling conventions, and



(a) Dynamic Instruction Counts



(b) Dynamic Memory References

Figure 5: Effects of our transformation for localizing globals/statics on program performance

whether the arguments are passed in registers or in memory (stack). All these added instructions can impose performance overheads for the transformed programs. At the same time, the compiler often generates more efficient code for accessing locals/arguments as compared to referencing globals. Also, most compilers are more aggressive at assigning locals/arguments to registers. Such effects can also improve program performance with our transformation.

We use the SimpleScalar set of functional simulators for the ARM platform [9] to derive accurate dynamic statistics for a *single-threaded* run of these benchmarks before and after our transformation.[2] All binary executables are fully optimized by VPO. We only collect the dynamic measurements for the six MiBench programs. Collecting dynamic results for most SPEC benchmarks is currently impractical due to their long simulation times. Moreover, imprecise call-graphs that are currently created by our compiler in the presence of indirect function calls for many of the SPEC CINT benchmarks result in incorrect code being produced for some of these programs after our transformation.[3]

Figure 5 presents the performance cost of our transformation in terms of dynamic instruction counts and dynamic number of load/store instructions for the MiBench programs. Our results reveal that while localizing a small number (and size) of static and global variables has minimal effect on performance, the overhead can become significant for benchmarks (like *ispell*) with large number and size of such variables. At the same time, conversion of globals to local variables can improve performance for benchmarks (like *dijkstra*) with comparatively small initialization and argument passing costs. Thus, implementing analysis algorithms to accurately identify and restrict the number of variables to localize is critical to the feasibility of this approach for larger benchmarks. Additionally, new optimization techniques may also help to manage the single-thread performance loss (for example, by eliminating initialization instructions if the variables are guaranteed to be overwritten before use, by combining multiple arguments that are always/mostly passed as a unit in a structure, and by more efficiently initializing blocks of storage by using a function such as *mmap*).

6. FUTURE WORK

There are a number of improvements that are necessary to address the limitations of our existing implementation, and increase the performance and attractiveness of the presented transformation. First, we plan to implement pointer analysis and improve alias analysis in VPO. Pointer analysis is necessary to 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. Second, by considering `main` as the thread-entry function allowed our existing implementation to ignore the possible cases where the closest common dominator for a global variable is *before* the threads are spawned. Our future implementation will include techniques to handle such cases. Third, we will implement enhanced analysis schemes to correctly identify the read/write status of static and global variables to provide more accurate information to the user and correctly localize only the necessary variables. Fourth, further research is needed to investigate the causes of performance overhead and develop optimizations to address them. Fifth, this transformation requires the user to accurately characterize globals and statics in the original code to generate its correct semantically equivalent thread-safe version. Further research will be needed to provide users with a more intuitive and precise user interface. Finally, we also plan to perform case-studies that involve parallelizing real programs to validate the feasibility of our interactive framework and assess the benefit of our technique during program parallelization efforts.

7. CONCLUSIONS

As the era of multi/many-core processors dawns on the computing community, it is becoming increasingly crucial to build parallel applications and convert existing sequential programs to use multiple cores. Automatic software techniques and tools have the potential of considerably easing this formidable software engineering task. The use of global and static variables produces non-thread-safe code, preventing their use in multi-threaded programs. Our results show the widespread use of such variables in existing programs, and motivate the importance of providing users with tools to appropriately handle them in parallel codes. In this paper we present a semi-automatic and interactive compiler-based approach to convert global and static variables in parallel programs into thread-safe storage. Unlike any existing solutions, our novel approach is completely compatible with current language, library and runtime systems. We provide a simple implementation of our transformation to primarily explore necessary analysis capabilities, its functional and performance characteristics, and identify avenues for performance improvements. From this study, we conclude that accurate implementation of our transformation is possible and is likely to enormously benefit program parallelization efforts to generate programs that remain compatible with existing software systems.

[2]Performance with multiple threads and on multicore machines will depend on the scalability achieved by the parallel program implementation.

[3]We manually verified that none of the functions invoked indirectly in the benchmark *jpeg* use static or global variables.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grant CNS-0953268.

9. REFERENCES

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

[2] International technology roadmap for semiconductors. accessed from http://www.itrs.net/Links/2008ITRS/Home2008.htm, 2008.

[3] Boost c++ libraries, boost 1.42.0 library documentation (chapter 21). Published at http://www.boost.org/doc/libs/1_42_0, February 2010.

[4] Msdn library, using thread local storage. Windows Developer Center, February 2010.

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

[6] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338. ACM Press, 1988.

[7] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Workshop on Languages and Compilers for Parallel Computing*, pages 10–1, 1994.

[8] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, 2007.

[9] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[10] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing, 1997.

[11] B. Carlson and S. Jahnke. Leveraging the benefits of symmetric multiprocessing (smp) in mobile devices. Texas Instruments white paper, 2009.

[12] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[13] M. Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001.

[14] U. Drepper. Elf handling for thread-local storage. Red Hat Inc., people.redhat.com/drepper/tls.pdf, December 2005.

[15] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, 2002.

[16] M. P. I. Forum. Mpi2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. Pvm: Parallel virtual machine – a users guide and tutorial for network parallel computing. MIT Press, 1994.

[18] D. Grove and L. Torczon. Interprocedural constant propagation: a study of jump function implementation. In *Proceedings of the 1993 conference on Programming language design and implementation*, pages 90–99, 1993.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[20] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, 2005.

[21] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2002.

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

[23] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The high performance Fortran handbook*. MIT Press, Cambridge, MA, USA, 1994.

[24] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, and Y. Paek. Vista: Vpo interactive system for tuning applications. *Transactions on Embedded Computing Systems*, 5(4):819–863, 2006.

[25] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[26] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Language Systems*, 1(1):121–141, 1979.

[27] R. Lischner. *Delphi in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, 2000.

[28] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 1 edition, 2008.

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

[30] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[31] K. H. Randall. Cilk: Efficient multithreaded computing. Technical report, Cambridge, MA, USA, 1998.

[32] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, Nov. 1997.

[33] R. M. Stallman and G. DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[34] W. R. Systems. Realize the promise of multi-core. http://www.windriver.com/announces/do-more-with-less/, 2011.

[35] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[36] K. Yelick, L. Semenzato, G. Pike, C. M. B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, December 1998.