Compiler Transformations to Generate Reentrant C Programs to Assist Software Parallelization

Adam R. Smith

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

Thesis Committee:

Dr. Prasad Kulkarni: Chairperson

Dr. Perry Alexander

Dr. Andy Gill

Date Defended

The Thesis Committee for Adam R. Smith certifies
That this is the approved version of the following thesis:

Compiler Transformations to Generate Reentrant C Programs to Assist Software Parallelization

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Acknowledgements

I thank my advisor, Professor Prasad Kulkarni, for being my sole well of knowledge throughout the development of this project. He seemed to always have the answer to any question that may have arose.

I am grateful to my fellow researchers, Michael Jantz and Manjiri Namjoshi, who helped me stay sane as we worked seven days a week.

I thank my friend, Jim Stevens, for introducing me to QEMU, which has saved me countless hours of simulation time.

I am forever indebted to my parents for all of their love and support that has allowed me to see that I can accomplish anything I set my mind to.

Most of all, I thank my fiancée, Miranda Brown, for her understanding and patience as she and my schoolwork vied for my time.

Abstract

As we move through the multi-core era into the many-core era it becomes obvious that thread-based programming is here to stay. This trend in the development of general purpose hardware is augmented by the fact that while writing sequential programs is considered a non-trivial task, writing parallel applications to take advantage of the advances in the number of cores in a processor severely complicates the process. Writing parallel applications requires programs and functions to be reentrant. Therefore, we cannot use globals and statics. However, globals and statics are useful in certain contexts. Globals allow an easy programming mechanism to share data between several functions. Statics provide the only mechanism of data hiding in C for variables that are global in scope.

Writing parallel programs restricts users from using globals and statics in their programs, as doing so would make the program non-reentrant. Moreover, there is a large existing legacy code base of sequential programs that are non-reentrant, since they rely on statics and globals. Several of these sequential programs display significant amounts of data parallelism by operating on independent chunks of input data, and therefore can be easily converted into parallel versions to exploit multi-core processors. Indeed, several such programs have been manually converted into parallel versions. However, manually eliminating all globals and statics to make the program reentrant is tedious, time-consuming, and error-prone. In this paper we describe a system to provide a semi-automated mechanism for users to still be able to use statics and globals in their programs, and to let the compiler automatically convert them into their semantically-equivalent reentrant versions enabling their parallelization later.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The *power wall*, which is a limit to the amount of power that a microprocessor chip can dissipate, has proven to be an impregnable barrier to the further scaling of microprocessor clock frequencies, effectively ending the era of high-performance uni-processor chips. Thus, software developers can no longer rely on perpetually increasing clock speeds to provide performance improvement for all programs. However, Moore's law of exponentially reducing feature (transistor) sizes is still maintaining its course. Processor manufacturers now exploit this continuous growth in transistor count to place multiple processing *cores*[1] on the same chip. Multi-core microprocessors employing two[2], four[3], and eight[4] cores are already shipping in general purpose machines, and many-core processors that employ tens[5] to hundreds[6] of cores can be purchased for high end server installa-

[1] In this context a single core can be thought of as an independent processing element, or CPU.

[2] Intel Core 2 Duo, AMD Athlon 64

[3] Intel Core i7, AMD Phenom

[4] Intel Nehalem-EX is an 8 core chip with 2 concurrent threads per core

[5] Azul Systems Vega 3 is a 54 core chip

[6] Intel Nehalem-EX and Azul Systems Vega 3 can be combined in multiple chip configurations with up to 128 and 864 concurrent threads, respectively

tions.

Multi-core and multi-processor based computers are poised to become the wave of the future. Programs on such machines derive their performance by executing multiple *threads* of execution concurrently on separate cores or processors. The demise of the uni-processor era was due in large part to microprocessors having hit the *power wall*, which is a limit to the amount of power that a microprocessor chip can dissipate. In the past, each successive generation of uni-processors consumed more power as they improved their performance by increasing their clock frequency. High clock rates result in higher thermal dissipation with potential overheating of the processor chip. Multi-core chips are an attempt to increase overall chip performance, without the need to scale the clock frequency of individual cores. Indeed, the scaling up of the number of cores per chip may come at the expense of scaling down the clock frequency of individual cores to maintain the power budget. Increasing the total number of cores on a chip and reducing clock speed can enable the processor to do more work in a given amount of time given that the processor keeps all cores busy. Thus, exploiting the available performance in multi-core chips requires programs to adopt the multi-threaded model of execution.

Unfortunately, the availability of multiple cores can provide no performance benefit to single-threaded applications. Consequently, software developers find themselves in an overdrive situation on two fronts: (a) generate new multi-threaded applications, and (b) convert existing sequential application to use multiple threads. It is clear that only scalable parallel applications will be able to gain continuous performance improvements on future processors.

While developing sequential applications is considered to be a non-trivial task,

writing multi-threaded applications has been found to increase the cost and complexity of software development. At the same time, parallel programming requires a different way of approaching the problem, and a different way of coding the solution. In particular, parallel programs need to be *reentrant*. A function is reentrant if, while it is being executed by one thread, it can be re-invoked by the same thread, or a routine in any other thread of the application. Thus, as far as parallel programs are concerned, reentrancy entails that it should be possible for multiple threads of execution to be simultaneously active in the same functions. Reentrancy requires all program state to be clearly partitioned into thread-specific (local) and shared (global) data. Modification of its thread-specific information by any thread should be invisible to all other threads. Additionally, in most cases, modifications to shared information must be atomic. Thus, developing parallel programs necessitates the elimination of *global* and *static* variables for all thread specific data, and the protection of global data from simultaneous access by multiple threads (typically by using mutexes or semaphores).

However, a significant number of legacy sequential C applications make generous use of static and global variables. We would also like to point out that several sequential programs exhibit significant coarse-grained data-level parallelism. Given the right tools, it appears relatively straight-forward to manually parallelize such applications to achieve scalable performance on multi-core machines. The problem therein lies in the burden placed on the user to parallelize the program.

Part of the extra burden that is placed on the user while writing parallel programs includes verifying that the function that the programmer is threadifying is reentrant. Non-reentrant functions can lead to hidden bugs in the code that appear only in certain circumstances, or may change depending on random system

events, such as race conditions, deadlock and livelock. This results in errors that, due to their inherent randomness, are difficult, if not impossible, to reproduce, which can greatly increase the amount of time and expense required to develop the application.

Automatically handling static and global variables to be reentrant will significantly reduce the user burden to parallelize such applications. At the same time, several programmers migrating from writing sequential applications for single-core processors to developing parallel programs for multi-core machines experience a steep learning curve before adapting to this change in their programming style. We believe that requiring programmers to stop using global and static variables in their programs may be unnecessary, if they could be automatically managed by the compiler—thereby easing the process of writing parallel applications. Moreover, several programming idioms can be naturally and more efficiently expressed using globals and statics. Statics also provide a very convenient mechanism of data hiding in non-object-oriented languages such as C, and even in C++.

In this thesis, we describe the issues involved in the implementation of a new program transformation to automatically eliminate static and global variables from a program to make it reentrant with minimal input from the user, thus easing the process of developing a parallel application. In certain instances, that must be specified by the user, the global data can be moved from the global data portion of memory, where it would be accessible to all threads, into the local memory for each thread. The transformation then automatically updates various function definitions with the number of arguments passed to each function to correctly reflect passing the new local variables to the necessary function. Once each thread has its own copy of the data, the user no longer needs to worry about

each thread's access to the data since the program is now reentrant, thereby alleviating the need for the programmer to do so by hand.

The rest of this thesis is organized as follows. Chapter 2 provides the reader with a basic understanding of globals and statics in C, reentrancy and basic terminology that is used throughout the text. Chapter 3 acknowledges related work in the area. Chapter 4 provides the basic structure of the compiler, the intermediate form the compiler uses, the environment in which data is collected and an introduction to the benchmarks we use in this thesis. Chapter 5 describes the process the transformation uses to make a program reentrant and how the intermediate code changes during transformation process. Chapter 6 discusses the benchmarks in detail, including the number and types of globals in the benchmarks and any overhead that the transformation may introduce. Chapter 8 provides concluding remarks and Chapter 7 discusses the road-map of this work as is continues into the future.

Chapter 2

Background

2.1 Globals, Statics and Their Implementation in C

In this section, we present a brief primer on the purpose and use of global variables in C, C++ and derived languages, describe their typical implementation and storage in the process address space, and discuss why the presence of global variables in a sequential program may make it more tedious to parallelize the application.

2.1.1 Implementation and Use of Global Variables

Any variable that is declared outside of the declaration of any function is said to have global scope. This means that the variable's data will persist throughout the execution of the program. These variable are typically referred to as *globals* in C terminology. Any global in a C program may be read or updated by any function in the application, including those that are not in the same file as the global.

An example program using a global can be seen in Figure 2.1. As mentioned

```
#include <stdio.h>

int glob;

void foo() {
    printf("%d\n", glob);
}

int main() {
    for (glob=0; glob<10; glob++)
        foo();
}
```

Figure 2.1. Example program with a global

previously, the value of global `glob` will be maintained throughout the entire execution of the program and the modifications of `glob` in `main()` will be visible in `foo()`. This technique, which should generally be avoided, also has the side-effect of reducing code size and increase performance in an application where the global may need to be accessed by many functions in the code. Indeed, many embedded and high performance application adopt this technique.

2.1.2 Implementation and Use of Static Variables

The *static* specifier can be used during the declaration of variables to extend the life of such variables to the entire program execution. A variable with a static specifier is commonly referred to as a *static* in C and C++ parlance. Thus, statics in C have the same global lifetime as globals. In spite of their global lifetime, static variables enjoy restricted visibility and scope that depends on where that static has been declared. A static declared outside a function definition has its scope restricted to the file in which it is declared, which means that the compiler will not allow this static variable to be accessed by any function not in that particular file.

```
int glob = 10;
static int filestat;

int proc() {
    int procloc;
    foo();
}

foo() {
    static int foostat = 0;
    int fooloc = 0;
    if (foostat < 10) {
        foostat++;
        foo();
    }
}
```

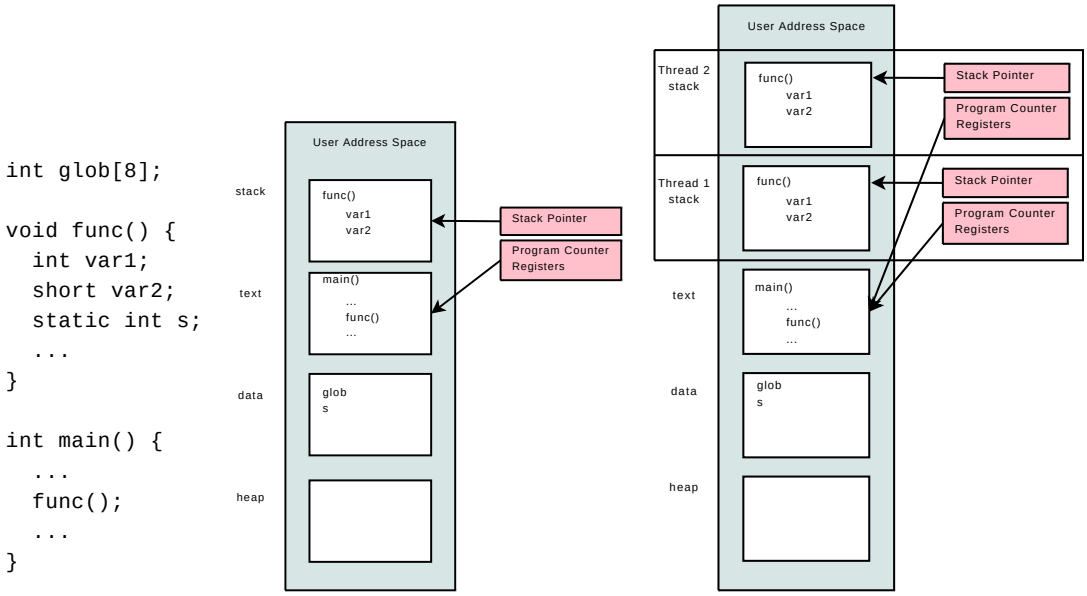
Figure 2.2. Declaration and use of static variables

Moreover, if the static variable is declared within some subroutine, then it will only be visible in that single subroutine. However, the static will retain its value between calls, so the function can preserve its state. Thus, the static specifier provides a very basic, yet convenient, mechanism to hide the visibility of global variables to only those places where they are needed.

Usage of static variables can be seen in Figure 2.2. The global variable `glob` will be visible to all functions in the entire program while the variable `filestat` will have its scope restricted to just the file in which it is declared. The variable `procloc` is local to procedure `proc` and thus will not retain its value across function calls. Variables `foostat` and `fooloc` will both have their scope limited to function `foo()`, but `foostat`'s value will persist across function calls. This means that every time function `foo()` is entered, a new copy of `fooloc` will be set to 0 as in its declaration line and will be destroyed upon exiting the function, but `foostat` will keep whatever previous value it had from the last time `foo()` was called.

2.1.3 Global Data in C

Due to their global lifetimes, the compiler detects all global or static variables during program compilation and allocates them space in the global data area of the process address space. The allocation of variables defined in the program illustrated in Figure 2.3(a) is shown in Figure 2.3(b). The typical process address space consists of four regions: code region (text region), data region, the heap, and the stack. *Local* variables (such as `var1` and `var2` in Figure 2.3(b)) reside in the *activation record* of each function on the stack. We can thus have multiple instances of the same local variable on the stack at the same time, one instance for each invocation of its enclosing function. In contrast, global variables (such as `glob` and `s` in program 2.3(a)) receive just one location in the data region for the entire program lifetime.



(a) Single Threaded Program Code (b) Single threaded program (c) Two threaded program

Figure 2.3. Example program with its address space

It is this single global allocation status of global variables that is responsible for the issues it creates for function reentrancy. Figure 2.3(c) shows a snapshot of the process address space for a multi-threaded version of the program in Figure 2.3(a), where the active threads, Thread 1 and Thread 2, have each just begun execution of `func()`. Both the threads get their own activation records on the stack where local data is stored, by virtue of which, there is no conflict between the local variables of the two threads. However, both threads share the same unique copy of static and global variables, creating conflicts if both threads are simultaneously accessing the same global. An obvious strategy to address this issue would be to remove the global from the global data area, and assign it storage in the stack region of each active thread, thus allowing each thread to have its own unique copy. This strategy is precisely the solution we adopt in this work to automatically result in reentrant C and C++ programs. However, this strategy is not always as straightforward to handle all categories of globals (discussed in chapter 5.1.1) due to their access patterns and compiler implementation issues. Notice also that function reentrancy is not only relevant for multi-threaded programs, but is also important for recursive functions and signal handlers.

2.2 What It Means to be Reentrant

We mentioned previously that a subroutine must be reentrant for it to be safely executed concurrently. In order for a a subroutine to be reentrant it must:

1. Hold no global non-constant data.

2. Not return the address of global non-constant data.

3. Work only on the data provided to it by the caller.

4. Not rely on locks to singleton resources.

5. Not modify its own code.

6. Not call non-reentrant computer programs or subroutines.

We can use compiler transformations to guarantee a certain subset of these conditions that is described in Section 5.1. The remaining conditions require intimate knowledge of the code that only the user can posses and therefore requires a certain amount of interaction to be sure that they are met.

2.3 Terminology

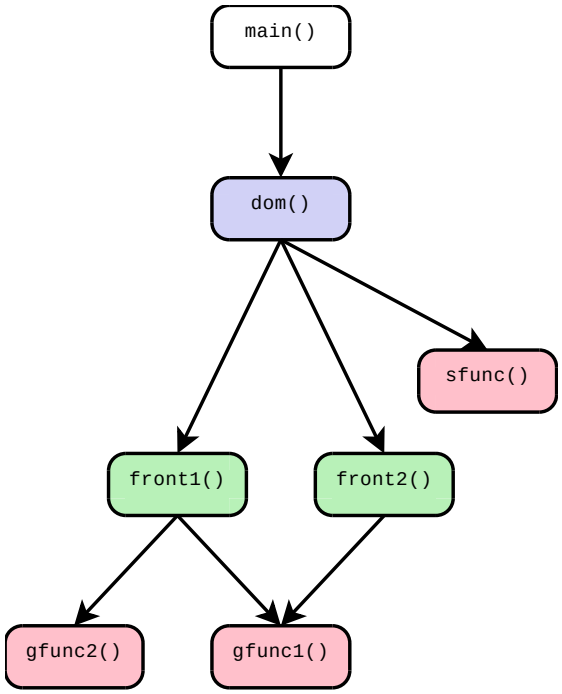
Included here is a brief overview of some of the terms we use to describe various parts of an application that is made reentrant. These terms will be used frequently throughout the rest of the text. All terms here refer to one or more nodes in a call graph. We refer to Figure 2.4 to define the following terms:

2.3.0.1 Global Function

A **global function** is any function that uses a global variable at any time during its execution. A **static function** is any function that uses a static variable. Throughout the text, we may use the term *global function* to refer to any function that uses data that resides in the global data area, i.e., it may be a static function *or* a global function. While slightly confusing, it makes it much easier to generalize. E.g., in order to make an entire program reentrant we must make all global functions reentrant. Indeed, when referring to a *global* we mean any variable whose data resides in the global data area, i.e., a static or global. We

```
int g;

int gfunc1() {
    return g++;
}

int gfunc2() {
    return g--;
}

int sfunc() {
    static int s;
    return s++;
}

int front1() {
    return gfunc1() + gfunc2();
}

int front2() {
    return gfunc1();
}

int dom() {
    return front1() + front2() + sfunc();
}

int main() {
    dom();
}
```

(a) Code



(b) Call Graph

Figure 2.4. Example call graph

take special care to point out specific static or global differences as they arise. In Figure 2.4, functions `gfunc1()`, `gfunc2()` and `sfunc()` are all *global functions*.

2.3.0.2 Function Dominator

A function d is said to dominate a function f if every path from the root of the call graph to f must go through d. Figure 2.4(a) shows an example program and its corresponding static call graph in Figure 2.4(b). Thus, in Figure 2.4, functions `main()`, `dom()` and `gfunc1()` all dominate function `gfunc1()`. A func-

tion d *strictly dominates* a function f if d dominates f and d does not equal f. Thus, in Figure 2.4, functions `main()` and `dom()` both strictly dominate function `gfunc1()`. The *immediate dominator*, or *idom*, of a function f is the unique node that strictly dominates f but does not strictly dominate any other node that strictly dominates f. Therefore every function has exactly one immediate dominator and in Figure 2.4, `dom()` is the immediate dominator of function `gfunc1()`.

A **static dominator**, for a particular static variable, is the function that is the *immediate dominator* of the function that declares and uses the static. This is fine and well for a static since a static can only be used in a single function, but a global can be used in any number of functions. Thus a **global dominator**, for some global variable, has two requirements: (a) it strictly dominates all functions that read or update that global and (b) it does not strictly dominate any function that meets requirement (a). Once again, there can be only one global dominator for a global function. In Figure 2.4, global `g` is used in functions `gfunc1()` and `gfunc2()`. Function `front1()` is the immediate dominator of function `gfunc2()` and function `dom()` is the immediate dominator of function `gfunc1()`. Therefore `dom()` is the *global dominator* of global `g`. The static function, `sfunc()`, shows the simplest case. Since `sfunc()` only has one parent, `dom()` is the static dominator for static `s`.

2.3.0.3 Function Frontier

A **static frontier** is the frontier of functions between a static function and its dominator. A **global frontier** is the frontier of functions between every global function of a global and that global's dominator. In Figure 2.4, the global frontier for global `g` is made up of the functions `front1()` and `front2()`. Since static

function `sfunc()`'s dominator is its parent, there are no functions between it and its dominator. Therefore, there are no functions in static `s`'s static frontier.

Chapter 3

Related Works

Writing parallel programs is universally acknowledged to be more complicated and error-prone than writing sequential code. Researchers and industry practitioners have developed numerous languages, compilers and tools to make it easier to write parallel programs, and to automatically transform sequential programs and loops to run concurrently on multi-processor and vector machines. In this section we briefly discuss work related to making it easier to develop parallel programs.

Many researchers have focused on new programming languages or extensions to existing programming languages to enable easier specification of parallelism [7, 10, 15, 21, 22, 27]. Current parallel programming languages, in many cases, present significantly different programming interfaces than their sequential counterparts, making their use unattractive for most traditional programmers. Traditional low-level multi-threading libraries, such as *Pthreads*, allow the programmer to express parallelism, but leave program and data partitioning, and synchronization, communication and deadlock management in the hands of the programmer [18]. Concurrent programming has also been long practised in the scientific

computing domain using data parallel language extensions and message passing libraries [11,12]. Our work does not propose a new parallel language, but a compiler transformation that can be employed during manual or automatic parallelization of traditional C and C++ programs.

Automatic compiler-based parallelization of sequential programs also has a rich history. Such approaches have achieved considerable success on regular scientific applications, by extracting DOALL parallelism [14] from loop nests accessing arrays [4, 26]. Unfortunately, automatic techniques have not been able to derive comparable results on general-purpose applications. Moreover, automatic parallelization is most commonly conducted using intraprocedural analysis within tight inner loops to exploit fine-grained and vector parallelism. Inter-procedural analysis required for coarse-grained parallelism is generally considered more complicated, and has only achieved limited success [13]. We are not aware of any automated inter-procedural technique that attempts to globally modify the function declarations, as is required to privatize global and static variables to make functions reentrant.

Attempts at integrating manual and automatic parallelization of applications have achieved interesting results. SUIF Explorer [20] enables programmer specification of parallelism guided by several tools to indicate parallelization opportunities and ensure correctness. The Software Behavior-Oriented Parallelization system also allows the programmer to indicate intended parallelism [8]. The tool attempts to make speculative parallelization more effective by employing user intuition and some other techniques, such as critical-path minimization and value based correctness checking. Tools to ease the exploitation of pipeline parallelism for streaming applications that exhibit regular data flows have been developed,

which also work in conjunction with a user [25]. Our work intends to use user knowledge to indicate the category of the global or static variable, as explained in Section 5.1.1.

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 [5]. Our handling of such class of global and static variables is based on a similar notion. However, this earlier work did not present details on the compiler implementation of their *Commutative* directive. Rinard et al. developed a novel analysis technique, called commutativity analysis, to automatically detect operations or functions that generate the same final result regardless of the order in which they execute [23]. Such analysis can be employed in our framework to supplement the manual annotation of the class of globals and statics that occur in commutative functions. Moreover, in addition to handling such commutative globals and statics, we are also interested in addressing the more challenging case of automatically managing globals and statics that do not share any information between concurrent threads of execution.

Our technique of localizing global and static variable accesses is most similar to the popular compiler optimizations of scalar expansion and scalar privatization [14]. 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 of *global replacement* outlined in this paper, scalar

expansion and privatization are local transformations and do not affect the calling interface of the function.

Chapter 4

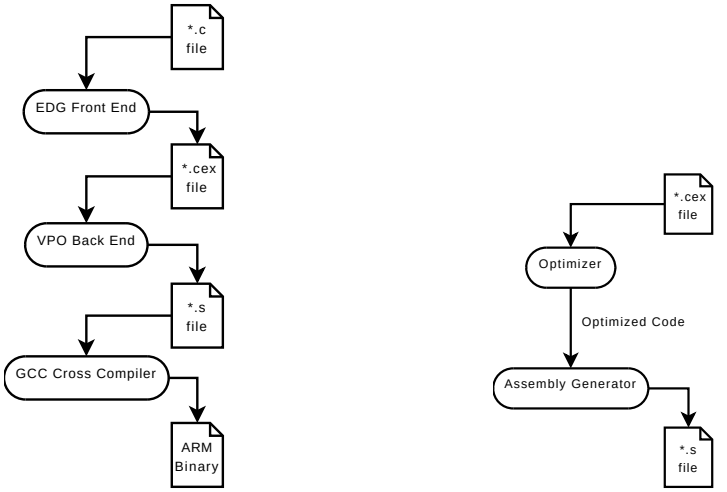
Design & Framework

4.1 Flow of Compiler

In this section we describe the compiler framework that was used throughout this project. At the highest level we can describe the flow of the program as it passes from a *.c source file into an executable. This process can be seen in Figure 4.1. Program compilation is composed of three major components as seen in Figure 4.1(a). The front end produces a list-based three-address intermediate code that is passed to the compiler back end that generates assembly. The assembly code produced by the compiler is then input to the assembler and linker that produce a machine executable binary.

We use the Edison Design Group's (EDG) C++ based compiler front end [9] that generates a *.cex file containing the intermediate representation of the original C source file. The *.cex file that is generated contains intermediate instructions in the form of Register Transfer Lists (RTLs) [3] that are used throughout the compilation process as the intermediate form. RTLs are described in Section 4.2.

The back end selected is the Very Portable Optimizer (VPO) [2, 3]. VPO



(a) Compiler Flow

(b) VPO Back End Flow

Figure 4.1. Flow graph of compiler

reads the intermediate code from the *.cex file, optimizes it and then generates an assembly file, as seen in Figure 4.1(b). As the name implies, the back end is very portable and supports several architectures. We have developed an RTL-to-RTL transformation that can be used to either output a new reentrant *.cex file, or produce assembly code for a particular architecture. We conducted our experiments on the StrongARM SA-100 platform. We use the compiler framework as a cross-compiler executing on x86 machines while generating code for the ARM.

Once the assembly files are obtained, they need to be assembled and linked. We use the GNU Compiler Collection (GCC) assembler and linker for this task. Once the user has obtained the executable, they are free to use it as they wish. We verified our executables on the platforms described in Section 4.3.

4.2 RTLs and the Intermediate Form

As mentioned previously, VPO reads in a *.cex file that contains intermediate code in the form of Register Transfer Lists (RTLs). This section provides a brief introduction to RTLs and how they are used. RTLs are a low-level, machine and language independent representation that encode machine-specific instructions. RTLs enable VPO optimizations to be largely implemented in a machine-independent fashion, yet allow easy migration of VPO to new platforms. Each RTL can be directly mapped to a single machine instruction. An example of RTLs can be seen in Figure 4.2. This is a very simple example to increment the value in `r[1]` (register 1).

4.2.0.4 Memory Accesses on the ARM

VPO provides five unique memory reference types for the ARM platform. The different types depend on the size and type of the variable being read. These memory access types are presented in Table 4.1. An example of reading a value can be seen in Figure 4.3(a) and an example of writing a value can be seen in Figure 4.3(b). The memory accesses denoted with B, W, and R will read 1, 2 or 4 bytes into an integer register, respectively, and the memory accesses denoted with F and D will read 4 or 8 bytes into a floating point register, respectively. Detailed examples of accessing global data can be seen in Appendix A.

```
r[1]=r[1]+1;
```

Figure 4.2. An example RTL

```
r[1]=R[r[2]];
```

(a) RTL to read 4 bytes from the address stored in `r[2]`

```
W[r[1]]=r[2];
```

(b) RTL to write 2 bytes to the address stored in `r[1]`

Figure 4.3. Example RTLs to access memory

Table 4.1. Memory access types

Access Type	Bytes Accessed	Data Type
B	1 Byte	char
W	2 Bytes	short
R	4 Bytes	int, long int, long long int
F	4 Bytes	float
D	8 Bytes	double, long double

4.3 Execution Environment

We employ VPO as a cross-compiler running on x86 machines and targeting the ARM architecture. We then use an ARM simulator to verify the correctness of the generated code. We use two different platforms for this task: The QEMU [1] virtual machine and the SimpleScalar ARM Simulator [6].

QEMU is a fast virtual machine that is capable of emulating many different architectures. We, naturally, used the ARM version. QEMU was chosen for its ability to quickly run generated code and verify its correctness.

The SimpleScalar ARM simulator is a cycle-accurate simulator that is also used to collect execution data on generated code. We found it to be significantly slower than QEMU, but it allows gathering valuable execution time information such as the number of machine cycles executed, cache and branch access information, etc., on two different executables. Thus, SimpleScalar allows us to compare the number of cycles taken to execute the original code with the number of cycles required to execute our modified code.

4.4 Benchmarks

We used the Standard Performance Evaluation Corporation's (SPEC) CPU2006 C Integer benchmarks to get various statistics on our transformation. The SPEC CPU2006 C Integer benchmarks [24] are a well known collection of software that provides a standard testing environment for hardware and compilers. The benchmarks are designed to test the performance of various hardware and compiler algorithms, however, they are all single-threaded, thus, are unable to take advantage of current multi-core systems. As previously mentioned, this is a problem that plagues many legacy applications. Indeed, the benchmarks have evolved over time from older software bases. Thus, they provide us with an ideal set of software to test our transformation. The integer benchmarks included in this suite are presented in Figure 4.2.

Table 4.2. The CINT2006 benchmarks

Benchmark	**Lines of Code**	**Description**
400.perlbench	155,497	The PERL programming language
401.bzip2	8,293	Lossless data compression
403.gcc	517,642	C compiler
429.mcf	2,685	Combinatorial optimization
445.gobmk	192,283	Artificial Intelligence: Go
456.hmmer	35,992	Search gene sequence
458.sjeng	12,846	Artificial Intelligence: chess
462.libquantum	4,357	Physics: quantum computing
464.h264ref	51,523	Video compression

Chapter 5

Implementation

In this chapter we describe our approach to semi-automatically transform a non-reentrant program into its semantically equivalent reentrant counterpart. The wide flexibility in program specification allowed by C (such as pointers, aliasing, loose typing, etc.) enables very efficient low-level program implementation, but prevents us from providing a completely general and automatic implementation.

5.1 Overview

5.1.1 Different Types of Globals and Statics

There are 3 different types of uses of globals as they pertain to parallel algorithms:

1. The globals that are completely thread-specific and independent of each other.

2. The global is shared between the threads, but the values need not be in any particular order.

3. Shared globals that must be accessed in the same order as in the original sequential program.

The user needs to specify what category each global falls into. Currently we are unable to do so. See Section 5.5 for a discussion on this.

We concentrate on categories 1 and 2. For category 1, we use our basic approach, as will be described in the next section. For category 2, we can synchronize access to global using mutexes or semaphores. This can easily be added in the RTLs. Category 3 presents an entirely different problem. The threads are dependent on their ordering, so they cannot be executed concurrently anyway. This code is difficult to parallelize without changing the algorithm. Thus, we cannot handle category 3 globals and the user must either handle them on his own, or change the algorithm.

Since we currently have no way of determining the category of globals, they are all assumed to be of category 1. This means that the user must still determine the category of each global on his own and manually handle category 2 and 3 globals.

5.1.1.1 Scope of Our Transformation

An example of a non-reentrant program and its semantically equivalent reentrant version can be seen in Figure 5.1. This is a fictional compiler for illustrative purposes only, although its architecture resembles that of VPO. In this compiler, each function in a file is compiled one at a time. The function is read from the input file, optimized and then written to the output file. Since the compilation of each function is entirely independent of every other function, this is a perfect candidate for parallelization. Thus, we create a thread for each function that

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

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

void compile() {
    read_next_func();
    optimize();
    write_opt_func();
}

void read_next_func() {
    /* set fn by reading in_fp */
    get_func_from_file();
}

void optimize() {
    /* optimize function
    pointed to by global fn */
}

void write_opt_func() {
    /* write out func pointed
    to by fn to out_fp */
}
```

(a) Non-reentrant compiler

```
FILE *in_fp;
FILE *out_fp;

int main() {
    while (more_functions) {
        compile_threads();
    }
}

void compile_thread() {
    create_thread(&compile);
}

void compile() {
    struct FN *fn;
    read_next_func(&fn);
    optimize(&fn);
    write_opt_func(&fn);
}

void read_next_func(FILE **fn) {
    /* lock file in_fp */
    get_func_from_file(fn);
    /* unlock file in_fp */
}

void optimize (FILE **fn) {
    /* optimize function pointed
       to by local fn */
}

void write_opt_func(FILE **fn) {
    /* lock file out_fp */
    /* write out func pointed
       to by fn */
    /* unlock file out_fp */
}
```

(b) Reentrant compiler

Figure 5.1. Example of making a program reentrant

needs to be compiled!

Before applying our compiler transformation, the user is responsible for determining how to best threadify the program. In this example, the function `compile()` is chosen, by the user, to be the entry point for each thread, as seen

in Figure 5.1(b).

In the non-reentrant version of the code, seen in Figure 5.1(a), there are three globals `fn`, `in_fp` and `out_fp`. `fn` is a type 1 global. Thus, each thread that will be created should get its own copy of `fn`, which will not be shared across threads in any way, since each will be working on a unique function. The globals `in_fp` and `out_fp` are of type 2. Each thread will need access to the file that is being read, but it needs to be ensured that each thread reads a unique function to compile. The reentrant version of the program accomplishes this by locking access to the file to guarantee that only one thread can be reading the file at any given time, thus ensuring that the file will be read sequentially and each thread will read a unique function. Similarly, access to the output file, accessed through `out_fp`, needs to be coordinated to ensure that the threads are not concurrently writing to the same file.

Our implementation, as described in the next sections, will be able to make the program reentrant, while the user is responsible for determining how best to parallelize the program, and implement the parallelization.

5.1.2 General Approach

Our basic approach to making a function reentrant is to first identify all definitions and uses of all statics or globals in the program. Then, we determine the dominator[1] of the function(s) that defines or uses the global or static. The storage for the global is then moved from the global data storage into the local data storage in the dominator function, as described in Section 2.1.1. The global is passed by reference to any function in the global's frontier. Thus, if the domi-

[1]More about function dominators can be found in Sections 2.3.0.2 and 5.3

nator is reentered, each thread will have its own copy of the "global" data that it can modify and not affect other threads of execution.

5.1.3 Example of Making a Program Reentrant

An example of a non-reentrant program and its semantically equivalent reentrant version produced by our transformation is illustrated in Figure 5.2. The technique to make the program reentrant follows our basic approach outlined above. First, for the non-reentrant program in Figure 5.2(a), identify all the global functions. In this case, our only global function is `gfunc()`. Then, we have to find the global dominator of each global. In this case, `dom()` is the global dominator of global `glob`. This is where the storage for `glob` will be placed, thus moving it from the global address space into the local address space for its global dominator. Now that `glob` no longer has global scope, we need to allow `gfunc()` to still be able to reference the data. This is accomplished by passing `glob` by reference to `gfunc()` and changing the accesses of `glob` to reflect these changes. The fully reentrant version can be seen in Figure 5.2(b). Now if `dom()` is reentered, each instance will have its own copy of the data to work with.

5.2 Flow Graph

5.2.1 VPO

We modified VPO to include our reentrification transformation before the code is optimized. Our transformation phase to make the program reentrant operates on the unoptimized program instance, and is inserted in the VPO compiler design flow as illustrated in Figure 5.3(a). An expanded view of the major components of our reentrification transformation can be seen in Figure 5.3(b). The optimization

```
int glob=0;

void dom() {
    gfunc();
}

void gfunc() {
    glob++;
}
```

(a) Non-reetrant

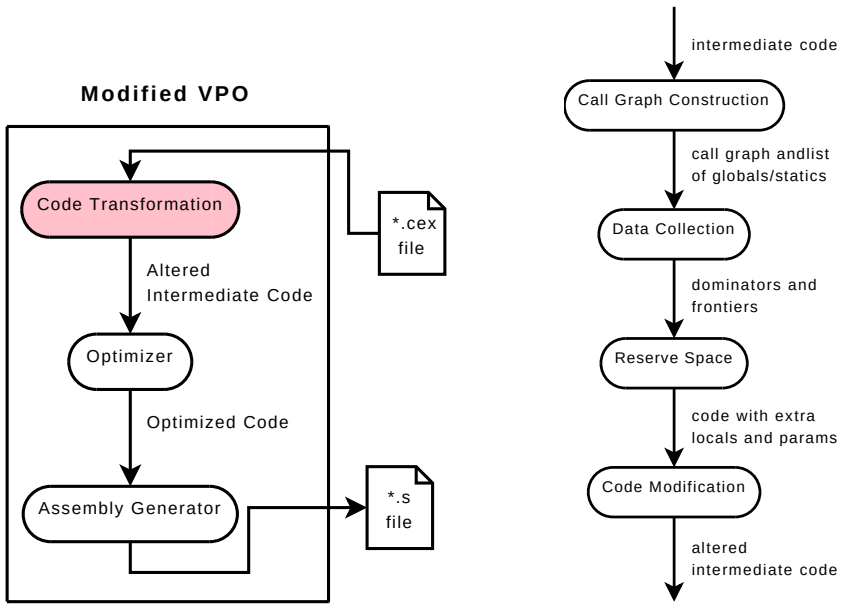
```
void dom() {
    int glob=0;
    gfunc(&glob);
}

void gfunc(int *glob) {
    *glob++;
}
```

(b) Reentrant

Figure 5.2. Example of a non-reentrant program with its semantically equivalent reetrant version

and assembly generation phases of VPO work in exactly the same way as they previously did and are unmodified. We also include a feature to turn the transformation on or off. If the transformation is off, VPO control flows in exactly the same way as before. This allows us to directly compare the effects of our transformation on the generated code.

There are two major advantages to applying our transformation before the optimizations are applied in VPO. First, we are guaranteed that the front end will always generate the same sequence of instructions for each global access, whereas after optimization the instruction sequences may be different. The predictable format of instructions in the unoptimized code makes it easy to identify global references, whereas the same reference instructions may be obscured in unpredictable ways after applying the optimizations. Secondly, it allows our changes to be optimized. Indeed, there are certain instructions that we need to replace with a new combination of instructions, but have left the old instructions in there that will be optimized away. This helps with debugging when we are looking at



(a) Modified VPO Flow (b) Transformation Flow

Figure 5.3. Modified compiler flow graph with changes shown in red

either the intermediate code or the assembly to see that the correct instructions are inserted in the correct place. More about this will be explained in Section 5.4.

5.2.2 Semi-Automatic Code Transformation

The transformation that semi-automatically makes a C program reentrant is broken down into four primary phases:

1. Call Graph Construction

   (a) Construct call graph

   (b) Build global and static list data structures

   (c) Note the function each global or static is used in

   (d) Store the initial values of each global or static

2. Data Collection From Call Graph

    (a) Prune call graph orphans

    (b) Remove unused globals or statics

    (c) Remove statics in main since main can't be reentered

    (d) Find function dominators

    (e) Find global dominators

    (f) Find global frontiers

    (g) Find memory access types

3. Reserving space for new locals and parameters

    (a) Add new parameters to global functions and their frontier functions

    (b) Add storage for global data as a local in the global dominators

4. Modification of RTLs to be Reentrant - just before optimization

    (a) Change global functions, intermediate functions and dominator functions

This process is shown in Figure 5.3(b). The first two phases of the transformation are both data collection phases, while the third and fourth phases actually change the structure of the RTLs with the information that it has collected. While the third phase adds additional memory usage overhead to the code by allocating space on the stack of each global dominator and the global frontier for new locals and parameters, respectively, only the fourth phase changes the semantics of the program.

5.2.2.1 Generation of the Call Graph

VPO already had a system to generate a call graph for a file, but several significant modifications had to be made. First and foremost, the compiler must be able to generate a complete call graph for the entire application. Since VPO was designed to work on a single file at a time, it had to be changed to be able to accept all necessary program files at the same time. It is important to be able to generate a complete call graph for the entire application for program semantics to be maintained consistent after our transformation. Thus, VPO is now able to take in all of the *.cex files corresponding to the original source files and generate the complete call graph from them in the call graph generation pass. The compilation pass still works on one function at a time and outputs the corresponding assembly code to the correct *.s file.

During the call graph phase, the call graph is generated by looking at each of the RTLs in the intermediate code and constructing nodes and edges as necessary. As mentioned previously, the transformation must make multiple passes over the intermediate code in order to collect the read information. During the call graph generation phase, the data that is collected includes: the call graph, a data structure containing each global found as well as their initial values, and the function(s) that use each global. This means that after this phase we now have a list of every global with its initial value and the function it is used in, i.e., the global functions.

5.2.2.2 Collection of Data From Call Graph

Our transformation then analyzes the call graph generated in the previous step to retrieve the required information before modifications to the program.

Steps that are needed in order to change the intermediate code to make it reentrant include:

Pruning call graph orphans. The call graph data structure contains a node for every function that appears in the source code as well as relevant information such as functions called by each node and functions that call that node. However, not every function that appears in the source code actually has a path from the `main()` function. This results in multiple extraneous nodes or graphs that cannot be reached when tracing from the root of the tree (the `main()` function) to each of the leaf nodes. We refer to these extraneous nodes as orphans, since they have no parents in the call graph. These orphan functions must be removed from the call graph in order for certain algorithms to work properly on it later.

Ignoring unused globals. We also found globals that are declared, and even initialized, but never referenced. This can happen for many reasons, including the programmer not checking for this, or compiler options. Thus, some globals that are declared at the top of a file and are only used in a certain subset of the functions that depend on a compiler option that was not enabled during a particular compilation, resulting in them never being referenced. We ignore such globals to ensure proper execution of the transformation.

Ignoring read-only globals. Sometimes globals are read-only. These globals do not affect reentrancy and can be ignored. Indeed, ignoring these globals will prevent any extra overhead that the transformation may introduce when handling globals.

Ignoring statics in main. Statics that appear in the `main()` function have to be removed for a couple of reasons. First, `main()` does not have any parents and thus any static used in it cannot have a static dominator. Secondly, statics used in `main()` can safely be ignored since `main()` cannot be reentered. Globals that are used in `main()` present a different problem since they can be used in other functions as well. This problem will be discussed in Section 5.2.2.4.

Finding function dominators. We update each node in the call graph data structure with a field containing that node's immediate dominator. Finding the immediate dominators is discussed in Section 5.3. The immediate dominator information is essential to our algorithm to find global dominators.

Finding global dominators. The global dominator for a particular global is the function in which the global is assigned storage after our transformation as a local. We need to know, for each global, the function in which to place this storage.

Finding global frontiers. Once you know all of the global functions and global dominators you can then find the global frontiers. This is any function in the call graph between the global function and its dominator.

Finding memory access types. The RTLs must be scanned to determine the type of memory access that is used when referencing a global. This needs to be determined before the modification of the RTLs so that you can use the correct memory access to read the data of the global. The intermediate code distinguishes between reading of floating point numbers and integers as well as reading 1, 2, 4 or 8 bytes.

5.2.2.3 Reserving Space for New Locals and Parameters

While VPO is populating the data structures for the optimization phase, we need to put some of the data that we collected to good use by reserving space for the new locals and parameters that will be added. In particular, this is the point that we add the new parameters to the global functions and frontiers. These parameters will store the address of each global that they need. In other words, the globals are passed by reference. There will also be space for the locals reserved in the global dominators's activation records to store the data for the global.

5.2.2.4 Modification of Intermediate Code

The final phase of the transformation is to change the RTLs to actually make the code reentrant. The first two phases of the transformation are strictly data collection phases. The third phase modifies the function prototypes of the global functions and frontier functions as well as making space for the data as locals in the respective global dominators, but does not actually change the semantics of the functions. After the third phase the code will use more memory (the newly added locals and parameters), but still executes exactly as it did before the changes. The fourth phase actually changes the intermediate code to make each function reentrant by using the new locals and parameters that were introduced in the third phase.

After the intermediate representation has been modified to be reentrant, it is then sent on to the optimizer. Thus, any changes we make to the application will still be optimized.

5.3 Global Dominators and Affected Functions

In this section, we describe the algorithm for identifying functions that will need to have their intermediate code changed in order to be made reentrant. We recognize three basic types of functions that are affected as part of our transformation to modify the intermediate code to make a program reentrant: (a) global functions, (b) global dominators, and (c) global frontiers. We identify each of the aforementioned functions as follows.

5.3.1 Global Functions

Global functions are, naturally, the easiest to identify as they are just the functions that use a global. These functions are detected during the generation of the call graph as the RTLs are read for the first time. Note that the pattern of RTLs that access a global remains the same in the unoptimized code, and hence can be easily and uniformly identified. An example pattern of RTLs accessing a global is shown in Figure 5.5(a) and will be discussed in Section 5.4.

5.3.2 Global Dominators

There are three different types of dominators that we care about. The first is just the immediate dominator of a function as described in Section 2.3.0.2. We used the Lengauer-Tarjan algorithm [19] to find these dominators. Since a static can only be used in a single function, the static dominators are simply the immediate dominator of any static function.

The global dominators are more difficult to determine, since a global may be used in any number of functions. When finding the immediate dominators of all of the functions, we saved them in the data structure by adding a link from the

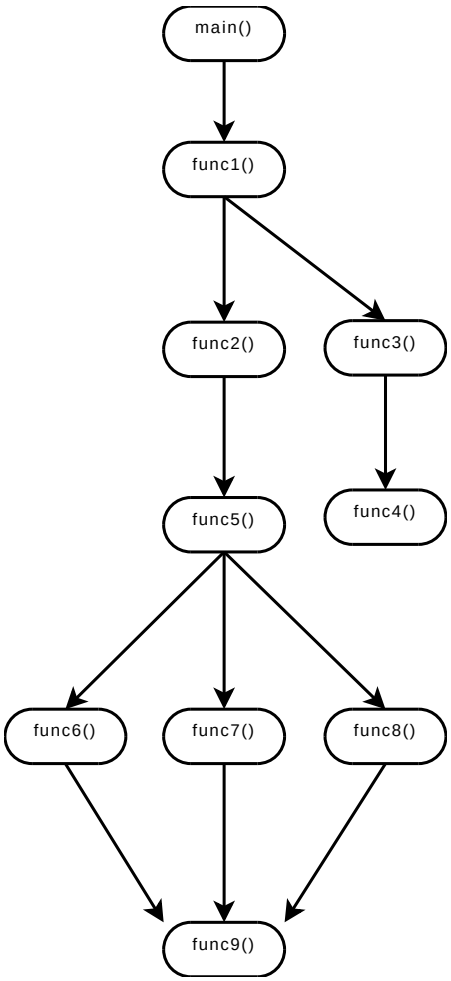
function to its immediate dominator. The global dominator can then be found by finding the first common ancestor of all of the functions that use the global by tracing up the call graph from immediate dominator to immediate dominator. For example, consider Figures 5.4(a) and (b) that show a sample program and its corresponding call graph, respectively. The global variable `g` is used in function `func4()` and `func9()`. Our algorithm first determines the immediate dominator of each use of the global functions, yielding `func5()` as the immediate dominator of the function `func9()` and func3() as the immediate dominator of the global used in `func4()`. We then determine the first common ancestor of all these immediate dominators, yielding `func1()` as the global dominator of global `g`. This is where the local definition of global `g` will be added later. The RTLs corresponding to this program are presented in Appendix A.2.

5.3.3 Global Frontiers

The Global Frontiers are much easier to calculate. For each global, a list of its global frontiers is maintained in a data structure. The frontier functions are found by starting at each global dominator adding all of its children to the frontier list. Then, the transformation adds all of the children of each of the functions that just were added and continues until it reaches each global function for that particular global. Now the list will contain all functions that reside in the call graph between the the global's global functions and the global's dominator. Thus, functions `func2()`, `func3()`, `func5()`, `func6()`, `func7()`, and `func8()` are the frontier functions for global `g`. After the local definition of `g` is moved to `func1()` (say as `loc_g`), we need to pass `loc_g` by reference as an argument to all frontier and global functions.

```
int glob = 2;

int func1() {
    return func2() + func3();
}

int func2() {
    return func5();
}

int func3() {
    return func4();
}

int func4() {
    glob += 1;
    return glob;
}

int func5() {
    return func6() + func7() + func8();
}

int func6() {
    return func9();
}

int func7() {
    return func9();
}

int func8() {
    return func9();
}

int func9() {
    glob -= 1;
    return glob;
}

int main() {
    return func1();
}
```

(a) Example program



(b) Call graph

Figure 5.4. Example program and its corresponding call graph

5.4 Intermediate Code and How It Changes

In VPO, the unoptimized function instance loads each global before its use, and stores it back after modification. The RTLs used to load or store the global are consistent, which allows us to easily update each reference to use the parameter that has been passed to the global function instead of the original global. See Appendix A for detailed code examples.

The RTLs used to reference a global in the unoptimized code can be seen in Figure 5.5(a) In this example, `r[14]` is a general purpose register and `r[13]` is the stack pointer. On line 1, `r[14]` is loaded with the value 4 that represents the offset on the stack of the pointer to the global data. The RTL on line 2, obtains the address of the pointer to the global data by adding the offset in `r[14]` to the stack pointer and this address is now stored in `r[14]`. Line 3 is a register to register transfer of the contents of `r[14]` to `r[1]`. Line 4 is the dereference of the address and reading of 4 bytes. After the block of RTLs in Figure 5.5(a) are executed the address of the global will be stored in `r[1]`.

Invariably, all accesses to a global are performed using the RTLs shown in Figure 5.5(a). Only the offset used will changed depending on which function is

```
1) r[14]=4;
2) r[14]=r[13]+r[14];
3) r[1]=r[14];
4) r[1]=R[r[1]];
```

(a) RTLs to reference a global.

```
1) r[14]=4;
2) r[14]=r[13]+r[14];
3) r[1]=r[14];
4) r[14]=8;
5) r[14]=r[13]+r[14];
6) r[1]=r[14];
7) r[1]=R[r[1]];
```

(b) Modified RTLs to reference a global.

Figure 5.5. RTLs to reference a global

being transformed and the global that needs to be accessed. Now that we have obtained the address of the global that we are referencing any number of things may be done. The value of the global can be read, written to or you can use the address in any means, such as passing it by reference to a function, or any other action that you may need the address of the global for.

Modifying the RTLs to use data other than the original data located in the global data section of the code is a trivial task once you have the means to identify all of the places that globals are referenced. You may simply replace those instructions to load the address of the data that you have added. For us, this means replacing the address of the global data with the address of the parameter that is passed in to the global function. An example of this can be seen in Figure 5.5(b). In this case, the parameter that was added was at an offset of 8 from the stack pointer.

As you can see, we leave the original instructions in the RTLs for the sake of development. It allows us to more easily see in the RTLs and in the assembly that the correct instructions have indeed been added. The semantics are the same as removing the original instructions and replacing them with the added instructions. This is slightly wasteful, as the old instructions that will have no effect will be executed, slightly degrading performance, but they can easily be optimized out.

5.5 Implementation Issues

As mentioned in Section 5.1.1, there are three types of globals as they pertain to parallel algorithms. Given the different categories of globals, the compiler needs a way to determine the category of each global. This task must be performed by the user. Possible implementations include using source-level annotations, or

employing an interactive compilation framework, such as the VISTA framework built into VPO [16, 17, 28]. So, we assume some knowledge of the program being compiled.

Currently, our implementation is not able to handle a few types of globals. These are arrays of pointers, structs, and globals that are initialized to large numbers. Handling arrays of pointers and globals with big initial values are architecture dependent issues. In order to create arrays of pointers we must inject assembly into the RTLs. The big initial values present a problem given that you are limited to 12 bits for values in the RTLs. Any values larger than this requires inserting assembly into the RTLs. We have solutions to these problems and will implement them as time permits.

Also, it is important to note that the user is responsible for identifying the point in the code at which to create threads as well as changing the code to create the threads.

The transformation is dependent on being able to construct a call graph for the entire program statically. This means that there can be no function pointers used in the program. Function pointers prevent the ability to statically determine the call graph and are not supported by the transformation. The user is warned about this and then has the choice of either changing the program to not use function pointers, or manually determining that the function being called by a pointer is already reentrant.

Changing of the calling conventions has some interesting implications based on the particular computer architecture. On the ARM platform, you can pass up to four arguments to a function in registers. If a function requires more than four parameters, the additional parameters have to be placed on the stack by

the caller, so that they may be read off of the stack by the callee. In the third phase of the transformation, space is added to allow for new parameters to be passed. While calculating how much space will be needed for each function, we are able to calculate the offset from the stack pointer to the new parameter on the stack, should parameters need to be passed on the stack. This offset is saved in the data structure for the global. In other words, we now know, for every function in a global's frontier, its offset on the stack, if it is not being passed in a register. Furthermore, while constructing the call graph, data was saved that indicated the pre-transformation number of parameters that are passed to every function. If the number of parameters is less than four, the transformation begins placing parameters in the next available parameter register. If at any point, the four parameter registers are already allocated, new parameters are added at the pre-computed stack offsets.

As mentioned previously, globals that are used in `main()` present a special case. The function `main()` has no dominating functions, so the global cannot be moved to be a local in `main()`'s dominator. This special case was handled by moving the data storage for the global to be a new local in `main()` as well as adding another local to store the address of the data that now resides in `main()`. This sounds a bit cumbersome, but it allows us to keep our same framework in place for `main()`. It is safe to make the data storage a local in `main()` since `main()` cannot be reentered. The second local that was added is taking the place of the parameter that would normally be added to `main()`'s function prototype to pass the global in by reference. When initializing the new local data, the transformation also places the address of the data into this new local to be reference just like the new parameter would have.

Chapter 6

Results

In this chapter we present statistics on the number of globals and statics used in our various benchmark programs. We present insights into the use of globals, as well as describe the overheads of our transformation in terms of static number of instructions added during the various changes made by the transformation. We use the SPECINT2006 benchmark programs to collect our results.

6.1 CINT2006 Benchmark Statistics

Table 6.1 shows the number of functions, globals and statics in each benchmark. Thus, we can see that the size of the benchmarks and the number of globals and statics in each benchmark varies greatly.

Table 6.2 shows the average number of functions that use each global in each benchmark. Since the scope of a static is limited to a single function, it can only be used in the function that defines it.

The counts of global and static access patterns can be seen in Tables 6.3 and 6.4, respectively. As mentioned previously, some of the benchmarks, 458.sjeng

Table 6.1. Total functions in the SPEC CINT2006 benchmarks

Benchmark	Total Functions	Num Globals	Num Statics
400.perlbench	607	207	2
401.bzip2	81	5	0
403.gcc	3890	696	50
429.mcf	23	4	0
445.gobmk	687	134	17
456.hmmer	207	6	16
458.sjeng	119	69	18
464.h264ref	503	215	14

Table 6.2. Average number of functions using each global

Benchmark	Average Number of Functions
400.perlbench	2.25
401.bzip2	1.00
403.gcc	2.64
429.mcf	1.00
445.gobmk	2.75
456.hmmer	2.00
458.sjeng	2.21
464.h264ref	3.20

in particular, makes heavy use of read-only globals or statics. For functions that are invoked often, allocating certain constant data (like the possible moves a chess piece can make from a given coordinate) in the global address space can prevent the function from initializing the data each time the function is called, thereby improving performance. By detecting read-only globals such as these and ignoring them, we can prevent our transformation from introducing any unnecessary overhead, since read-only globals don't affect reentrancy.

Our algorithm could not determine the access patterns of some global variables. The column "Unknown" in Tables 6.3 and 6.4 represents these globals. Investigation into this phenomenon revealed that such globals were only accessed by reference. Thus, the address of the global was loaded into a register, but that

Table 6.3. Global access patterns

Benchmark	Read-Only	Write-Only	Unknown	Read/Write
400.perlbench	38	18	6	145
401.bzip2	3	1	0	1
403.gcc	211	32	19	434
429.mcf	0	0	0	4
445.gobmk	51	7	2	74
456.hmmer	2	0	1	3
458.sjeng	30	4	0	35
464.h264ref	70	16	18	111

Table 6.4. Static access patterns

Benchmark	Read-Only	Write-Only	Unknown	Read/Write
400.perlbench	0	0	2	0
401.bzip2	0	0	0	0
403.gcc	11	0	8	31
429.mcf	0	0	0	0
445.gobmk	3	0	2	12
456.hmmer	1	0	1	14
458.sjeng	13	0	0	5
464.h264ref	9	0	0	5

address was neither directly read from or written to. Examples of what may happen to the address include aliasing the global data with another pointer or passing the global by reference to some function, such as `memset()`. In such cases our algorithm loses the ability to track what happens to the data in the global. These particular globals cannot be handled automatically. The user is presented with a warning describing this case and they are required to handle the global manually.

Table 6.5 presents the number of reentrant and non-reentrant functions in each benchmarks due to statics and globals. Thus, we can see that a large number of functions in each benchmark are non-reentrant, highlighting the importance of the transformation presented in this thesis. For example, in the GCC benchmark

Table 6.5. Reentrant and non-reentrant functions in the benchmarks *after* ignoring read-only globals

Benchmark	**Reentrant**	Non-Reentrant Due To		
		Global	**Static**	**Combined**
400.perlbench	117	490	301	490
401.bzip2	79	2	0	2
403.gcc	1040	2828	2550	2850
429.mcf	19	4	0	4
445.gobmk	214	472	187	473
456.hmmer	175	25	14	32
458.sjeng	43	76	2	76
464.h264ref	189	314	14	314

Table 6.6. Reentrant and non-reentrant functions in the benchmarks *before* ignoring read-only globals

Benchmark	**Reentrant**	Non-Reentrant Due To		
		Global	**Static**	**Combined**
400.perlbench	107	500	301	500
401.bzip2	64	17	0	17
403.gcc	868	3015	2552	3022
429.mcf	19	4	0	4
445.gobmk	205	481	187	482
456.hmmer	175	29	14	32
458.sjeng	34	78	42	85
464.h264ref	170	330	27	333

only 1,040 out of the total 3,890 functions are reentrant.

The data in Table 6.5 is collected *after* ignoring read-only and write-only globals. By ignoring read-only and write-only globals we were able to save some overhead in all benchmarks except for 429.mcf, which used very few globals to begin with. The number of reentrant and non-reentrant functions due to globals *before* ignoring read-only and write-only globals is presented in Table 6.6.

6.2 Single-Threaded Overhead Introduced by the Transformation

The goal of this research is to make it easier for a programmer to parallelize an application to take advantage of increasing number of cores on a chip, thereby increasing the application's performance. However, the astute reader will have noticed that many of the transformations we have performed may add instructions to the code, thereby decreasing single-threaded performance in some cases. However, the transformation is intended to enable the user to parallelize the application and run multiple threads concurrently. Thus, although our transformation may result in slightly lower per thread performance, with many threads executing in parallel, a total increase in overall performance is expected. The performance overhead introduced by the transformation is as follows.

Increased memory usage. After we move the global into the local storage in the global's dominator, every thread that is generated will have its own copy of the data. This means that the total memory usage of the application will increase each time a dominator function is entered. The total amount of memory used by the program cannot be calculated statically however, because, for example, a function may be recursive and reenter itself a variable number of times.

The average number of functions in a global or static's frontier is shown in Table 6.7. For some of the benchmarks, the average number of functions in a global's frontier is zero. For benchmarks with a non-zero number of global variables, the number of functions in its global frontier can still be zero if the benchmark only has a single parent, resulting in the parent being the global's dominator, thus leaving no functions between the global function and the global dominator.

Table 6.7. Number of functions in global and static frontiers

Benchmark	Global			Static		
	Min	**Max**	**Average**	**Min**	**Max**	**Average**
400.perlbench	0	490	324.17	3	490	246.50
401.bzip2	0	0	0.00	0	0	0.00
403.gcc	0	2500	215.14	0	1517	325.79
429.mcf	0	0	0.00	0	0	0.00
445.gobmk	0	407	61.70	0	174	48.94
456.hmmer	0	15	7.00	0	3	1.06
458.sjeng	0	29	7.08	0	0	0.00
464.h264ref	0	192	13.51	0	4	2.40

Some of the benchmarks that make heavy use of globals result in a large number of frontier functions, on average, for each global. The 403.gcc and 400.perlbench had particularly high numbers of functions in the average global frontier. We must point out that a user not using our semi-automatic transformation may have to perform this analysis on their own. They would have to determine the global's dominator as well as the frontier functions. Then they would have to painstakingly perform the accounting by hand and ensure that they passed the global as a parameter along until it reaches the destined global function. It is plain to see that this can get out of hand very quickly in terms of the introduced overhead.

One word of data is added to each frontier and global function for each global that they take as a new parameter. This cost may increase the memory usage slightly.

Increased number of instructions. We add instructions in three main places: in the global functions, in the frontier functions and in the dominator functions. Every time we add instructions we are not only increasing the number of instructions needed to be executed when the program is run, thus increasing execution

time, but also increasing code size. This section describes the overhead involved in adding instructions to make a program reentrant. Keep in mind that all overhead is on a per-thread basis, so when running multiple threads concurrently, overall performance is still expected to increase.

In each global function, the RTLs are changed to access the new parameter instead of the the global data, which introduces no additional instructions. In fact, we can actually reduce overhead in some cases. With the way that the RTLs are presently created in VPO it requires 2 temporary locals and 11 instructions, 5 of which are memory accesses, for the first time you access a global. The first temporary is initialized with 4 of the instructions, using 2 memory accesses. This places the address of a global table into the first temporary. The remaining 7 instructions, with 3 memory accesses, uses the previously computed table pointer to index into the table to find the address of the desired global. The second temporary now contains the address of this global. This overhead is introduced with the assumption that the global will be accessed many times in the function and instructions and memory access will be saved by keeping the address of the global in a temporary local. All of those instructions are no longer needed if we remove all globals from a function, thus saving this overhead.

The global frontier functions and the global dominator functions are also required to pass the address of the newly created local containing the global data along to the global function. As mentioned previously, on the ARM platform, up to four parameters can be passed in registers. Any more than that will have to be passed on the stack. If the function is able to pass the parameter in a register it requires three instructions with one memory access to do so. If there are already four parameters being passed in registers, then the function will have to use five

instructions with two memory accesses.

There are instructions that need to be added to initialize the new local variable inside of the global dominator functions. This requires three instructions with one memory access to initialize a scalar global. Currently, we treat arrays and structs as list of scalars, so if an array is declared as 1,000 elements, the transformation will include 3,000 instructions which increases code size and execution time. We are currently investigating efficient ways to bring this number down. One of the biggest problems in this respect is that uninitialized globals are zeroed out. This means that we have to introduce instructions to zero out the global when we move it to the local address space. If we do not do this, the programmer may assume that it is zeroed out in the code and we will produce incorrect code. Perhaps an efficient function call like `memset()` can be used in the future. Chapter 7 provides further discussion on this topic.

Chapter 7

Future Work

The VPO compiler has built in support for interacting with the user. The general framework of the transformation could be extended to use this interaction to gain knowledge about the globals. In particular, it is up to the user to provide knowledge about the type of each global. Currently, we assume that the user has already handled type 2 and 3 globals and only focus on type 1 globals. If VPO interactions were extended to include gathering information about global types, we could continue our work to handle type 2 globals. As mentioned previously, type 2 globals can be made reentrant by providing a locking mechanism, such as mutexes or semaphores to ensure that only one thread is accessing the global at a time. Furthermore, interaction with the user may very well provide them with more clues about how to handle each of the globals in the program. The data collected during the transformation will ensure that all of the globals are accounted for and presented to the user to prevent the user from accidentally skipping a global.

There are times that we might introduce a lot of instructions that will increase code size and execution time for each thread. In particular, we add instructions

to initialize uninitialized globals to zero. Uninitialized identifiers in the global address space are automatically zeroed for you. Thus, when we move the global into the local address space, we must explicitly initialize the data to zero as well to maintain the correctness of the code. Currently, this has the potential to add significant overhead into the system. We would like to find a way eliminate this overhead if possible. Perhaps we can implement a more efficient way to handle this case, or, through interaction with the user, not initialize the data if it is not needed.

Another potentially dangerous case that the transformation cannot currently handle is aliasing of the global data or passing the global by reference to a function. If the user does this, the transformation loses track of accesses to the global. We are looking into a method to determine when aliasing of the global data occurs to warn the user that the code may have potentially harmful effects. Currently we display a general warning if we cannot determine a global's access type, but any extra information will ease the user's job of handling this manually.

Further lessening of the overhead introduced by the transformation can be achieved through interaction with the user by determining the entry point for the threads. If the user discloses this information during the transformation, only those functions that are descendents of the function that the user specifies need to be made reentrant.

Chapter 8

Conclusion

In this thesis we discussed compiler transformations to semi-automatically generate reentrant C code. The parallelization of software is known to be a difficult problem, but it is necessary to take advantage of current and future general purpose hardware. Any work that can be done automatically, or semi-automatically, can help reduce developer burden and allow them to focus on the algorithms that they employ and not low level bookkeeping to make a program reentrant. Indeed, we have shown that there are many globals and statics used in common software, such as the SPEC CINT2006 benchmark suite.

Furthermore, our transformation allows the user to continue to use the well developed practice of using statics and globals. Of particular note is the ability to continue to use the static qualifier, which provides the only means of data hiding in C.

In summary, I believe that as current trends to shift software development to the multi-core and many-core domain, away from the well established sequential paradigm, that software development will continue to become increasingly complex. Take, for example, the heterogeneous parallel architectures such as GPGPU

or the Cell Broadband Engine Architecture. I strongly believe that programmers will begin to adopt automatic or semi-automatic tools to help them achieve the best performance possible. As we march into the future of many-core architectures, perhaps transformations, such as those outlined in this thesis, will lead the charge.

Appendix A

Code Examples

I have included some example code here showing a C program with its corresponding source file file, generated RTL file and assembly file.

A.1 Overview of RTL files

When looking at an RTL file it is important to note that the first character on every line describes the type of instruction that is contained on the line. For example, # denotes a comment line, + denotes an RTL line, and - denotes an assembly line. In certain instances extra comments will be added for clarity.

A.2 A Code Example

This example was discussed in Section 5.3. The call graph for this program can be seen in Figure 5.4 on page 38.

A.2.1 C File

```
int glob = 2;

int func1() {
return func2() + func3();
}

int func2() {
return func5();
}

int func3() {
return func4();
}

int func4() {
glob += 1;
return glob;
}

int func5() {
return func6() + func7() + func8();
}

int func6() {
return func9();
}

int func7() {
return func9();
}

int func8() {
return func9();
}

int func9() {
glob -= 1;
return glob;
}

int main() {
return func1();
}
```

A.2.2 Non-Reentrant RTLs

```
# Beginning of concatenation of all *.cex files
# Beginning file global_tree4.cex
-.globl func1
-.globl func2
-.globl func3
-.globl func4
-.globl func5
-.globl func6
-.globl func7
-.globl func8
-.globl func9
-.globl main
-.data
-.globl glob
-.data
-.align 2
-glob:
-.word 2
-.text
-.align 2
-.L1:
-.word .L2
-func1:
+r[32]=L1;
+r[32]=LA[r[32]];
+r[33]=r[11]+.TMP.0;
+R[r[33]]=r[32];
#     File global_tree4.c Line 16
#     File global_tree4.c Line 17
+ST=func2;
+ST=func3;
+r[34]=r[32]+r[33];
+r[0]=r[34];
+PC=RT;
-.data
-.align 2
-.L2:
-.text
-.align 2
-.L4:
-.word .L5
-func2:
+r[35]=L4;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.1;
+R[r[36]]=r[35];
#     File global_tree4.c Line 20
#     File global_tree4.c Line 21
+ST=func5;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L5:
-.text
-.align 2
-.L7:
-.word .L8
-func3:
+r[33]=L7;
```

```
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.2;
+R[r[34]]=r[33];
#     File global_tree4.c Line 24
#     File global_tree4.c Line 25
+ST=func4;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L8:
-.text
-.align 2
-.L10:
-.word .L11
-func4:
+r[33]=L10;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.3;
+R[r[34]]=r[33];
+r[33]=r[11]+.TMP.3;
+r[33]=R[r[33]];
+r[34]=0;
+r[33]=r[33]+r[34];
+r[35]=LA[r[33]];
+r[36]=r[11]+.TMP.4;
+R[r[36]]=r[35];
#     File global_tree4.c Line 28
#     File global_tree4.c Line 29
+r[32]=1;
+r[34]=r[11]+.TMP.4;
+r[33]=R[r[34]];
+r[35]=R[r[33]];
+r[37]=r[11]+.TMP.4;
+r[36]=R[r[37]];
+r[38]=r[35]+r[32];
+R[r[36]]=r[38];
+
#     File global_tree4.c Line 30
+r[33]=r[11]+.TMP.4;
+r[32]=R[r[33]];
+r[34]=R[r[32]];
+r[0]=r[34];
+PC=RT;
-.data
-.align 2
-.L11:
-.word glob
-.text
-.align 2
-.L13:
-.word .L14
-func5:
+r[35]=L13;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.5;
+R[r[36]]=r[35];
#     File global_tree4.c Line 33
#     File global_tree4.c Line 34
+ST=func6;
+ST=func7;
+r[34]=r[32]+r[33];
+ST=func8;
```

```
+r[36]=r[34]+r[35];
+r[0]=r[36];
+PC=RT;
-.data
-.align 2
-.L14:
-.text
-.align 2
-.L16:
-.word .L17
-func6:
+r[37]=L16;
+r[37]=LA[r[37]];
+r[38]=r[11]+.TMP.6;
+R[r[38]]=r[37];
#     File global_tree4.c Line 37
#     File global_tree4.c Line 38
+ST=func9;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L17:
-.text
-.align 2
-.L19:
-.word .L20
-func7:
+r[33]=L19;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.7;
+R[r[34]]=r[33];
#     File global_tree4.c Line 41
#     File global_tree4.c Line 42
+ST=func9;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L20:
-.text
-.align 2
-.L22:
-.word .L23
-func8:
+r[33]=L22;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.8;
+R[r[34]]=r[33];
#     File global_tree4.c Line 45
#     File global_tree4.c Line 46
+ST=func9;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L23:
-.text
-.align 2
-.L25:
-.word .L26
-func9:
+r[33]=L25;
```

```
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.9;
+R[r[34]]=r[33];
+r[33]=r[11]+.TMP.9;
+r[33]=R[r[33]];
+r[34]=0;
+r[33]=r[33]+r[34];
+r[35]=LA[r[33]];
+r[36]=r[11]+.TMP.10;
+R[r[36]]=r[35];
#     File global_tree4.c Line 49
#     File global_tree4.c Line 50
+r[32]=1;
+r[34]=r[11]+.TMP.10;
+r[33]=R[r[34]];
+r[35]=R[r[33]];
+r[37]=r[11]+.TMP.10;
+r[36]=R[r[37]];
+r[38]=r[35]-r[32];
+R[r[36]]=r[38];
+
#     File global_tree4.c Line 51
+r[33]=r[11]+.TMP.10;
+r[32]=R[r[33]];
+r[34]=R[r[32]];
+r[0]=r[34];
+PC=RT;
-.data
-.align 2
-.L26:
-.word glob
-.text
-.align 2
-.L28:
-.word .L29
-main:
+r[35]=L28;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.11;
+R[r[36]]=r[35];
#     File global_tree4.c Line 54
#     File global_tree4.c Line 55
+ST=func1;
+r[0]=r[32];
+PC=RT;
-.data
-.align 2
-.L29:
# End file global_tree4.cex
```

A.2.3 Reentrant RTLs

```
# Beginning of concatenation of all *.cex files
# Beginning file global_tree4.cex
-.globl func1
-.globl func2
-.globl func3
-.globl func4
-.globl func5
-.globl func6
-.globl func7
-.globl func8
-.globl func9
-.globl main
-.data
-.globl glob
-.data
-.align 2
-glob:
-.word 2
-.text
-.align 2
-.L1:
-.word .L2
-func1:
#
#Initializing new local for global glob
+r[35]=r[11]+.new_local_glob;
+r[36]=2;
+R[r[35]]=r[36];
#End initializing new local
#
+r[32]=L1;
+r[32]=LA[r[32]];
+r[33]=r[11]+.TMP.0;
+R[r[33]]=r[32];
#     File global_tree4.c Line 16
#     File global_tree4.c Line 17
#
#Passing param for global glob
+r[35]=r[11]+.new_local_glob;
+r[0]=r[35];
+ST=func2;
+
#End passing param
#
#
#Passing param for global glob
+r[35]=r[11]+.new_local_glob;
+r[0]=r[35];
+ST=func3;
+
#End passing param
#
+r[34]=r[32]+r[33];
+r[0]=r[34];
+PC=RT;
-.text
-.align 2
-.L4:
-.word .L5
-func2:
```

```
+r[35]=L4;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.1;
+R[r[36]]=r[35];
#     File global_tree4.c Line 20
#     File global_tree4.c Line 21
#
#Passing param for global
+r[37]=r[11]+.new_param_glob;
+r[37]=R[r[37]];
+r[0]=r[37];
+ST=func5;
+
#End passing param
#
+r[0]=r[32];
+PC=RT;
-.text
-.align 2
-.L7:
-.word .L8
-func3:
+r[33]=L7;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.2;
+R[r[34]]=r[33];
#     File global_tree4.c Line 24
#     File global_tree4.c Line 25
#
#Passing param for global
+r[35]=r[11]+.new_param_glob;
+r[35]=R[r[35]];
+r[0]=r[35];
+ST=func4;
+
#End passing param
#
+r[0]=r[32];
+PC=RT;
-.text
-.align 2
-.L10:
-.word .L11
-func4:
+r[33]=L10;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.3;
+R[r[34]]=r[33];
#
#Static/Global Declaration
+r[33]=r[11]+.TMP.3;
+r[33]=R[r[33]];
+r[34]=0;
+r[33]=r[33]+r[34];
+r[35]=LA[r[33]];
+r[36]=r[11]+.TMP.4;
+R[r[36]]=r[35];
#End Static/Global Declaration
#
#     File global_tree4.c Line 28
#     File global_tree4.c Line 29
+r[32]=1;
#
```

```
#Global Reference
+r[34]=r[11]+.TMP.4;
+r[34]=r[11]+.new_param_glob;
+r[33]=R[r[34]];
#End Global Reference
#
+r[35]=R[r[33]];
#
#Global Reference
+r[37]=r[11]+.TMP.4;
+r[37]=r[11]+.new_param_glob;
+r[36]=R[r[37]];
#End Global Reference
#
+r[38]=r[35]+r[32];
+R[r[36]]=r[38];
+
#     File global_tree4.c Line 30
#
#Global Reference
+r[33]=r[11]+.TMP.4;
+r[33]=r[11]+.new_param_glob;
+r[32]=R[r[33]];
#End Global Reference
#
+r[34]=R[r[32]];
+r[0]=r[34];
+PC=RT;
-.text
-.align 2
-.L13:
-.word .L14
-func5:
+r[35]=L13;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.5;
+R[r[36]]=r[35];
#     File global_tree4.c Line 33
#     File global_tree4.c Line 34
#
#Passing param for global
+r[37]=r[11]+.new_param_glob;
+r[37]=R[r[37]];
+r[0]=r[37];
+ST=func6;
+
#End passing param
#
#
#Passing param for global
+r[37]=r[11]+.new_param_glob;
+r[37]=R[r[37]];
+r[0]=r[37];
+ST=func7;
+
#End passing param
#
+r[34]=r[32]+r[33];
#
#Passing param for global
+r[37]=r[11]+.new_param_glob;
+r[37]=R[r[37]];
+r[0]=r[37];
```

```
+ST=func8;
+
#End passing param
#
+r[36]=r[34]+r[35];
+r[0]=r[36];
+PC=RT;
-.text
-.align 2
-.L16:
-.word .L17
-func6:
+r[37]=L16;
+r[37]=LA[r[37]];
+r[38]=r[11]+.TMP.6;
+R[r[38]]=r[37];
#     File global_tree4.c Line 37
#     File global_tree4.c Line 38
#
#Passing param for global
+r[39]=r[11]+.new_param_glob;
+r[39]=R[r[39]];
+r[0]=r[39];
+ST=func9;
+
#End passing param
#
+r[0]=r[32];
+PC=RT;
-.text
-.align 2
-.L19:
-.word .L20
-func7:
+r[33]=L19;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.7;
+R[r[34]]=r[33];
#     File global_tree4.c Line 41
#     File global_tree4.c Line 42
#
#Passing param for global
+r[35]=r[11]+.new_param_glob;
+r[35]=R[r[35]];
+r[0]=r[35];
+ST=func9;
+
#End passing param
#
+r[0]=r[32];
+PC=RT;
-.text
-.align 2
-.L22:
-.word .L23
-func8:
+r[33]=L22;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.8;
+R[r[34]]=r[33];
#     File global_tree4.c Line 45
#     File global_tree4.c Line 46
#
```

```
#Passing param for global
+r[35]=r[11]+.new_param_glob;
+r[35]=R[r[35]];
+r[0]=r[35];
+ST=func9;
+
#End passing param
#
+r[0]=r[32];
+PC=RT;
-.text
-.align 2
-.L25:
-.word .L26
-func9:
+r[33]=L25;
+r[33]=LA[r[33]];
+r[34]=r[11]+.TMP.9;
+R[r[34]]=r[33];
#
#Static/Global Declaration
+r[33]=r[11]+.TMP.9;
+r[33]=R[r[33]];
+r[34]=0;
+r[33]=r[33]+r[34];
+r[35]=LA[r[33]];
+r[36]=r[11]+.TMP.10;
+R[r[36]]=r[35];
#End Static/Global Declaration
#
#     File global_tree4.c Line 49
#     File global_tree4.c Line 50
+r[32]=1;
#
#Global Reference
+r[34]=r[11]+.TMP.10;
+r[34]=r[11]+.new_param_glob;
+r[33]=R[r[34]];
#End Global Reference
#
+r[35]=R[r[33]];
#
#Global Reference
+r[37]=r[11]+.TMP.10;
+r[37]=r[11]+.new_param_glob;
+r[36]=R[r[37]];
#End Global Reference
#
+r[38]=r[35]-r[32];
+R[r[36]]=r[38];
+
#     File global_tree4.c Line 51
#
#Global Reference
+r[33]=r[11]+.TMP.10;
+r[33]=r[11]+.new_param_glob;
+r[32]=R[r[33]];
#End Global Reference
#
+r[34]=R[r[32]];
+r[0]=r[34];
+PC=RT;
-.text
```

```
-.align 2
-.L28:
-.word .L29
-main:
+r[35]=L28;
+r[35]=LA[r[35]];
+r[36]=r[11]+.TMP.11;
+R[r[36]]=r[35];
#     File global_tree4.c Line 54
#     File global_tree4.c Line 55
+ST=func1;
+r[0]=r[32];
+PC=RT;
```

A.2.4 Non-Reentrant ARM Assembly

```
 @ reg_param_space=0, int_save_space=4
 @ local start=8
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 8
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 8
 @ fp is not necessary, using FP
 @  Beginning of concatenation of all *.cex files
 @  Beginning file global_tree4.cex
.globl func1
.globl func2
.globl func3
.globl func4
.globl func5
.globl func6
.globl func7
.globl func8
.globl func9
.globl main
.data
.globl glob
.data
.align 2
glob:
.word 2
.text
.align 2
.L1:
.word .L2
func1:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #12
.TMP.0 = 8
 @ end of fixentry
adr r12,.L1
ldr r12,[r12]
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .TMP.0
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 16
 @      File global_tree4.c Line 17
bl func2
str r0,[sp,#0]
bl func3
ldr r12,[sp,#0]
add r12,r12,r0
mov r0,r12
add sp, sp, #12
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L2:
 @ reg_param_space=0, int_save_space=4
```

```
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L4:
.word .L5
func2:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.1 = 0
 @ end of fixentry
adr r12,.L4
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.1
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 20
 @      File global_tree4.c Line 21
bl func5
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L5:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L7:
.word .L8
func3:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.2 = 0
 @ end of fixentry
adr r12,.L7
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.2
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 24
```

```
 @      File global_tree4.c Line 25
bl func4
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L8:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 8
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L10:
.word .L11
func4:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #8
.TMP.4 = 0
.TMP.3 = 4
 @ end of fixentry
adr r12,.L10
ldr r12,[r12]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.3
mov r0,lr
str r12,[r0]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.3
mov r12,lr
ldr r12,[r12]
mov r0,#0
add r12,r12,r0
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 28
 @      File global_tree4.c Line 29
mov r12,#1
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r0,lr
ldr r0,[r0]
ldr r0,[r0]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r1,lr
```

```
ldr r1,[r1]
add r12,r0,r12
str r12,[r1]
 @      File global_tree4.c Line 30
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r12,lr
ldr r12,[r12]
ldr r12,[r12]
mov r0,r12
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L11:
.word glob
 @ reg_param_space=0, int_save_space=4
 @ local start=8
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 8
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 8
 @ fp is not necessary, using FP
.text
.align 2
.L13:
.word .L14
func5:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #12
.TMP.5 = 8
 @ end of fixentry
adr r12,.L13
ldr r12,[r12]
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .TMP.5
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 33
 @      File global_tree4.c Line 34
bl func6
str r0,[sp,#0]
bl func7
ldr r12,[sp,#0]
add r12,r12,r0
str r12,[sp,#0]
bl func8
ldr r12,[sp,#0]
add r12,r12,r0
mov r0,r12
add sp, sp, #12
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
```

```
.L14:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L16:
.word .L17
func6:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.6 = 0
 @ end of fixentry
adr r12,.L16
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.6
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 37
 @      File global_tree4.c Line 38
bl func9
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L17:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L19:
.word .L20
func7:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.7 = 0
 @ end of fixentry
adr r12,.L19
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.7
mov r0,lr
```

```
str r12,[r0]
 @      File global_tree4.c Line 41
 @      File global_tree4.c Line 42
bl func9
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L20:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L22:
.word .L23
func8:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.8 = 0
 @ end of fixentry
adr r12,.L22
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.8
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 45
 @      File global_tree4.c Line 46
bl func9
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L23:
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 8
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L25:
.word .L26
func9:
```

```
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #8
.TMP.10 = 0
.TMP.9 = 4
 @ end of fixentry
adr r12,.L25
ldr r12,[r12]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.9
mov r0,lr
str r12,[r0]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.9
mov r12,lr
ldr r12,[r12]
mov r0,#0
add r12,r12,r0
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 49
 @      File global_tree4.c Line 50
mov r12,#1
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r0,lr
ldr r0,[r0]
ldr r0,[r0]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r1,lr
ldr r1,[r1]
sub r12,r0,r12
str r12,[r1]
 @      File global_tree4.c Line 51
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r12,lr
ldr r12,[r12]
ldr r12,[r12]
mov r0,r12
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L26:
.word glob
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
```

```
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L28:
.word .L29
main:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.11 = 0
 @ end of fixentry
adr r12,.L28
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.11
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 54
 @      File global_tree4.c Line 55
bl func1
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L29:
 @ fp is not necessary, using FP
 @ fp is not necessary, using FP
 @  End file global_tree4.cex
```

A.2.5 Reentrant ARM Assembly

```
 @ reg_param_space=0, int_save_space=4
 @ local start=8
 @ my_arg_build_size=0
 @ local size = 8
 @  spill size = 8
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 8
 @ fp is not necessary, using FP
 @  Beginning of concatenation of all *.cex files
 @  Beginning file global_tree4.cex
.globl func1
.globl func2
.globl func3
.globl func4
.globl func5
.globl func6
.globl func7
.globl func8
.globl func9
.globl main
.data
.globl glob
.data
.align 2
glob:
.word 2
.text
.align 2
.L1:
.word .L2
func1:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #16
.new_local_glob = 8
.TMP.0 = 12
 @ end of fixentry
 @
 @ Initializing new local for global glob
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_local_glob
mov r12,lr
mov r0,#2
str r0,[r12]
 @ End initializing new local
 @
adr r12,.L1
ldr r12,[r12]
mov lr,#12
add lr,sp,lr
 @  fixentry loaded .TMP.0
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 16
 @      File global_tree4.c Line 17
 @
 @ Passing param for global glob
mov lr,#8
```

```
add lr,sp,lr
 @  fixentry loaded .new_local_glob
mov r12,lr
mov r0,r12
bl func2
 @ End passing param
 @
 @
 @ Passing param for global glob
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_local_glob
mov r12,lr
str r0,[sp,#0]
mov r0,r12
bl func3
 @ End passing param
 @
ldr r12,[sp,#0]
add r12,r12,r0
mov r0,r12
add sp, sp, #16
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L2:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L4:
.word .L5
func2:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #4
.TMP.1 = 0
.new_param_glob = 4
 @ end of fixentry
adr r12,.L4
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.1
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 20
 @      File global_tree4.c Line 21
 @
 @ Passing param for global
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
```

```
ldr r12,[r12]
mov r0,r12
bl func5
 @ End passing param
 @
mov r0,r0
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L5:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L7:
.word .L8
func3:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #4
.TMP.2 = 0
.new_param_glob = 4
 @ end of fixentry
adr r12,.L7
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.2
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 24
 @      File global_tree4.c Line 25
 @
 @ Passing param for global
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
mov r0,r12
bl func4
 @ End passing param
 @
mov r0,r0
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L8:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
```

```
 @ my_arg_build_size=0
 @ local size = 8
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L10:
.word .L11
func4:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #8
.TMP.3 = 0
.TMP.4 = 4
.new_param_glob = 8
 @ end of fixentry
adr r12,.L10
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.3
mov r0,lr
str r12,[r0]
 @
 @ Static/Global Declaration
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.3
mov r12,lr
ldr r12,[r12]
mov r0,#0
add r12,r12,r0
ldr r12,[r12]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r0,lr
str r12,[r0]
 @ End Static/Global Declaration
 @
 @      File global_tree4.c Line 28
 @      File global_tree4.c Line 29
mov r12,#1
 @
 @ Global Reference
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r0,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r0,lr
ldr r0,[r0]
 @ End Global Reference
 @
ldr r0,[r0]
 @
 @ Global Reference
mov lr,#4
```

```
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r1,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r1,lr
ldr r1,[r1]
 @ End Global Reference
 @
add r12,r0,r12
str r12,[r1]
 @      File global_tree4.c Line 30
 @
 @ Global Reference
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.4
mov r12,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
 @ End Global Reference
 @
ldr r12,[r12]
mov r0,r12
add sp, sp, #12
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L11:
.word glob
 @ reg_param_space=0, int_save_space=8
 @ local start=8
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 8
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 8
 @ fp is not necessary, using FP
.text
.align 2
.L13:
.word .L14
func5:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #12
.TMP.5 = 8
.new_param_glob = 12
 @ end of fixentry
adr r12,.L13
ldr r12,[r12]
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .TMP.5
mov r0,lr
str r12,[r0]
```

```
 @      File global_tree4.c Line 33
 @      File global_tree4.c Line 34
 @
 @ Passing param for global
mov lr,#12
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
mov r0,r12
bl func6
 @ End passing param
 @
 @
 @ Passing param for global
mov lr,#12
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
str r0,[sp,#0]
mov r0,r12
bl func7
 @ End passing param
 @
ldr r12,[sp,#0]
add r0,r12,r0
 @
 @ Passing param for global
mov lr,#12
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
str r0,[sp,#0]
mov r0,r12
bl func8
 @ End passing param
 @
ldr r12,[sp,#0]
add r12,r12,r0
mov r0,r12
add sp, sp, #16
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L14:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L16:
.word .L17
func6:
```

```
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #4
.TMP.6 = 0
.new_param_glob = 4
 @ end of fixentry
adr r12,.L16
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.6
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 37
 @      File global_tree4.c Line 38
 @
 @ Passing param for global
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
mov r0,r12
bl func9
 @ End passing param
 @
mov r0,r0
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L17:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L19:
.word .L20
func7:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #4
.TMP.7 = 0
.new_param_glob = 4
 @ end of fixentry
adr r12,.L19
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.7
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 41
 @      File global_tree4.c Line 42
```

```
 @
 @ Passing param for global
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
mov r0,r12
bl func9
 @ End passing param
 @
mov r0,r0
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L20:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L22:
.word .L23
func8:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #4
.TMP.8 = 0
.new_param_glob = 4
 @ end of fixentry
adr r12,.L22
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.8
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 45
 @      File global_tree4.c Line 46
 @
 @ Passing param for global
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
mov r0,r12
bl func9
 @ End passing param
 @
mov r0,r0
add sp, sp, #8
ldmfd sp!, {pc}
mov r0,r0
```

```
 @ fp is not necessary, using FP
.data
.align 2
.L23:
 @ reg_param_space=0, int_save_space=8
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 8
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 8
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L25:
.word .L26
func9:
 @ func is not irq handler
stmfd sp!, {r0, lr}
sub sp, sp, #8
.TMP.9 = 0
.TMP.10 = 4
.new_param_glob = 8
 @ end of fixentry
adr r12,.L25
ldr r12,[r12]
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.9
mov r0,lr
str r12,[r0]
 @
 @ Static/Global Declaration
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.9
mov r12,lr
ldr r12,[r12]
mov r0,#0
add r12,r12,r0
ldr r12,[r12]
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r0,lr
str r12,[r0]
 @ End Static/Global Declaration
 @
 @      File global_tree4.c Line 49
 @      File global_tree4.c Line 50
mov r12,#1
 @
 @ Global Reference
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r0,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r0,lr
ldr r0,[r0]
```

```
 @ End Global Reference
 @
ldr r0,[r0]
 @
 @ Global Reference
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r1,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r1,lr
ldr r1,[r1]
 @ End Global Reference
 @
sub r12,r0,r12
str r12,[r1]
 @      File global_tree4.c Line 51
 @
 @ Global Reference
mov lr,#4
add lr,sp,lr
 @  fixentry loaded .TMP.10
mov r12,lr
mov lr,#8
add lr,sp,lr
 @  fixentry loaded .new_param_glob
mov r12,lr
ldr r12,[r12]
 @ End Global Reference
 @
ldr r12,[r12]
mov r0,r12
add sp, sp, #12
ldmfd sp!, {pc}
mov r0,r12
 @ fp is not necessary, using FP
.data
.align 2
.L26:
.word glob
 @ reg_param_space=0, int_save_space=4
 @ local start=0
 @ my_arg_build_size=0
 @ local size = 4
 @  spill size = 0
 @ calculate_non_scratch_int_reg_save_size = 4
 @ we will need the LR saved
 @  spill size = 0
 @ fp is not necessary, using FP
.text
.align 2
.L28:
.word .L29
main:
 @ func is not irq handler
stmfd sp!, {lr}
sub sp, sp, #4
.TMP.11 = 0
 @ end of fixentry
adr r12,.L28
ldr r12,[r12]
```

```
mov lr,#0
add lr,sp,lr
 @  fixentry loaded .TMP.11
mov r0,lr
str r12,[r0]
 @      File global_tree4.c Line 54
 @      File global_tree4.c Line 55
bl func1
mov r0,r0
add sp, sp, #4
ldmfd sp!, {pc}
mov r0,r0
 @ fp is not necessary, using FP
.data
.align 2
.L29:
 @ fp is not necessary, using FP
 @ fp is not necessary, using FP
 @  End file global_tree4.cex
```

References

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[2] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 329–338, New York, NY, USA, 1988. ACM.

[3] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the international conference on Programming languages and system architectures*, pages 105–124, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[4] 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 *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[5] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[8] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. *SIGPLAN Not.*, 42(6):223–234, 2007.

[9] Edison Design Group. `http://www.edg.com/`.

[10] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[12] 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.

[13] D. Grove and L. Torczon. Interprocedural constant propagation: a study of jump function implementation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 90–99, New York, NY, USA, 1993. ACM.

[14] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[15] 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.

[16] 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, Y. Paek, and D. Jones.

Vista: Vpo interactive system for tuning applications. volume 5, pages 819–863, November 2006.

[17] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23. ACM Press, 2003.

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

[19] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[20] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–48, New York, NY, USA, 1999. ACM.

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

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

[23] 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.

[24] SPEC CINT2006. `http://www.spec.org/cpu2006/CINT2006/`.

[25] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[26] 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.

[27] 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.

[28] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones. Vista: A system for interactive code improvement. In *ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 155–164. ACM, June 2002.