© 2006 by Isaac James Dooley. All rights reserved.

AUTOMATED SOURCE-TO-SOURCE TRANSLATIONS TO ASSIST PARALLEL PROGRAMMERS

BY

ISAAC JAMES DOOLEY

B.S., Birmingham-Southern College, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

It is commonly believed that parallel programming can be difficult. Because it is already difficult enough, any automatic ways of simplifying parallel programming are welcome. This thesis describes how source-to-source translators can be used to make parallel programming easier. This thesis describes three source-to-source translators and provide some examples of translated codes. The translators used the ROSE library.

The first source-to-source translator removes global and static variables from a program and encapsulates them in a structure which is allocated on the stack. This translator facilitates the use of any MPI code with AMPI, an adaptive MPI implementation which unfortunately has limited support for MPI applications containing global or static variables. The source-to-source translator just transforms the code a minimal amount so that existing compilers, linkers, and loaders can be used. Thus no special compilers or loaders are required. Requiring special compilers or loaders necessarily would limit the acceptance of AMPI, whereas a source-to-source translator would provide a simple automated way of transforming an MPI code for use on any platform.

The second source-to-source translator inserts calls to functions for tracing an application. The traces are used for post-mortem performance analysis. The translator inserts a Projections function tracing call at the beginning and end of each function in the source code. Thus an existing MPI application can be analyzed for performance function by function, without any need for the user to manually modify the application's code.

The third source-to-source translator automatically creates PUP routines for Charm++ applications. PUP routines are functions that serialize the state of a migratable object.

PUP routines are normally created by hand, which leaves room for programmer error. For example, it is easy to overlook a member variable in a class. The source-to-source translator, however, can easily iterate through all member variables and add each to the PUP routine.

AMPI shows great potential for improving the performance of scientific simulations that use MPI. AMPI adaptively manages parallel resources, provides runtime instrumented load balancing, fault-tolerance, and supports virtualization. These features are very useful for dynamic applications and are essential for large scale parallel codes. AMPI augmented with the global variable rewriting translator will be able to provide these features automatically to any C or C++ MPI application. Parallel scientific applications are generally concerned with two main criteria, capabilities and performance. To analyze performance, the source code instrumenting tool will be useful. All AMPI scientific application developers can benefit from this tool. These translation tools will increase the productivity of parallel programmers.

To Thomas and Laura Dooley,

My Loving Parents.

Acknowledgments

I would like to thank my advisor Laxmikant Kale, and all the members of the Parallel Programming Laboratory for their help with various aspects of this work. I would like to thank the HPCS Fellowship program which provided funding for this project.

Table of Contents

List of Tables . . . . . **ix**

List of Figures . . . . . **x**

List of Abbreviations . . . . . **xi**

Chapter 1 Introduction **1**
1.1 Related Work 2

Chapter 2 Source-To-Source Translation **4**
2.1 ROSE for Source-to-Source Translation 6
2.2 The Simplest Possible ROSE Translator 8
2.3 Complications 9
2.3.1 Languages 9
2.3.2 Compiler Flags 9
2.4 Stability and Robustness 10

Chapter 3 Thread Variable Privatization **11**
3.1 Introduction 11
3.2 Data Layout for Variables in C and C++ 12
3.2.1 Globals and Statics: An Overview 12
3.3 Global Variables with Virtualization 14
3.3.1 Manually Rewrite Source Code 15
3.3.2 Swap Globals 16
3.3.3 Compiler Based Solutions 16
3.3.4 Automatically Rewrite Source Code 17
3.3.5 Example translations 18
3.4 Case Study: MILC 19
3.4.1 MIMD Lattice Computations 19
3.5 Case Study: WRF 22
3.5.1 Description of WRF 22
3.5.2 Globals and Statics Variables in WRF 22

Chapter 4 Automated Performance Tracing 26
4.1 AMPI Performance Tracing with Projections 26
4.2 Performance Tracing Source-to-Source Translator 27
4.3 Example 28

Chapter 5 Automatic PUP Function Creation 29
5.1 PUP Functions 29
5.2 PUP Creator Source-to-Source translator 30
5.3 Example 31

Chapter 6 Example Translated Codes 32

Chapter 7 Conclusions 39

Appendix A Thread Variable Privatization: Code Listing 40

Appendix B Automated Performance Tracing: Code Listing 52

Appendix C Automatic PUP Function Creation: Code Listing 56

Appendix D Common Shared Routines: Code Listing 59

References 63

List of Tables

3.1 2 source-code files with global and static variables. 13

List of Figures

2.1 Source-to-source overview 5
2.2 A selection of the 241 AST nodes used by ROSE 7
2.3 A simple identity translator that uses ROSE 8

3.1 Memory layout for a program and a user-level threaded program 13
3.2 MPI example program with a global variable called `my_rank` 14
3.3 Code from figure 3.2 manually rewritten to eliminate the global variable . . . 15
3.4 Global Variables in compiled MILC object files 21
3.5 Count of Global BSS symbols in WRF executables 23
3.6 43 Global BSS symbols in WRF executable ideal.exe 24
3.7 61 Global BSS symbols in WRF executable wrf.exe 25

5.1 A C++ class with a PUP function 30

6.1 A C++ file before the automatic creation of PUP functions 32
6.2 A C++ file after the automatic creation of PUP functions 33
6.3 A sample program before tracing calls are inserted 34
6.4 Program of Figure 6.3 after tracing calls are inserted to a depth of 3. 35
6.5 After inserting tracing calls to a depth of 2 36
6.6 An example program with multiple global variables 37
6.7 Translated version of program from Figure 6.6. Contains no global variables. 37
6.8 Symbols created when compiling the code in Figure 6.6 37
6.9 Symbols created when compiling the code in Figure 6.7 37
6.10 An example program with two static variables. Both static variables are named `a`, but they have different scopes. 38
6.11 After translation, the two static variables are replaced with two variables with mangled names are in the struct. The end of the mangled name is the original name `a`. 38

List of Abbreviations

AST	Abstract Syntax Tree. A graph representation of source code
ELF	Executable and Linkable Format (formerly Extensible Linking Format)
GAS	Global Address Space
MILC	MIMD Lattice Computation
NAS	NASA Advanced Supercomputing Division
NASA	National Aeronautics and Space Administration
NCAR	National Center for Atmospheric Research
NPB	NAS Parallel Benchmark
PPL	Parallel Programming Lab
PUP	Pack and Un-Pack
QCD	Quantum Chromodynamics
ROSE	ROSE Source-to-Source Translation Library
WRF	Weather Research and Forecasting simulation developed by NCAR

Chapter 1

Introduction

It is commonly believed that parallel programming can be difficult. Because it is already difficult enough, any automatic ways of simplifying parallel programming are welcome. This thesis describes how source-to-source translators can be used to make parallel programming easier. This thesis describes three source-to-source translators in chapters 3, 4, and 5 while providing some examples of translated codes in chapter 6. The translators use the ROSE library which is described in chapter 2.

The first source-to-source translator removes global and static variables from a program and encapsulates them in a structure which is allocated on the stack. This translator facilitates the use of any MPI code with AMPI, an adaptive MPI implementation which unfortunately does not allow MPI applications to contain global or static variables on non 32-bit ELF platforms. The source-to-source translator just transforms the code so that existing compilers, linkers, and loaders can be used. Thus no special compilers or loaders are required for all relevant platforms. Requiring special compilers or loaders necessarily would limit the acceptance of AMPI, whereas a source-to-source translator would provide a simple automated way of transforming an existing MPI code for use with AMPI on any platform.

The second source-to-source translator inserts calls to functions for tracing an application. The traces are used for post-mortem performance analysis. The translator inserts function tracing call at the beginning and end of each function in the source code. Thus an existing MPI application can have its performance traced function by function in a reasonable manner, without any need for the user to manually modify the application's code. Just as many vendors provide tracing libraries relevant to their own particular machines, and as

many MPI implementations provide their own tracing facilities, AMPI and Charm++ use their own performance analysis tool Projections. Projections is a very advanced tool which is currently being extending to provide even more performance views for AMPI traced applications.

The third source-to-source translator automatically creates PUP routines for Charm++ applications. PUP routines are functions that serialize the state for a migratable object. PUP routines are normally created by hand, which leaves room for programmer error. For example, it is easy to overlook a member variable in a class. The source-to-source translator, however, can easily iterate through all member variables and add each to the PUP routine.

This thesis describes three source-to-source translators using the ROSE library along with their motivating issues and examples of translated codes. The examples are simple enough for the reader to quickly understand, but the translators have been used on much larger codes spanning dozens of files and thousands of lines of code. The translators are available in the Charm++ source code repository for any interested readers to view or use.

1.1 Related Work

There are no existing software source-to-source translation tools that perform the specific translations done by the three source-to-source tools described in this thesis. The translators proposed in this thesis use the ROSE library[1, 14, 16, 15]. Other frameworks could have been used to perform similar tasks, but ROSE best suited the initial goals of the project. For example, the LLVM compiler infrastructure[10, 9, 8] is a particularly well suited framework to build the translators described in this thesis, however it was not chosen because its generated output code does not look like the original C or C++ input. It is a goal of our work to be able to rewrite an application and retain its original structure as much as possible, including the comments which would have been lost in a LLVM based translation. ROSE just translates from source to source not to some optimized machine language as does

LLVM. Another research oriented extensible compiler infrastructure for source-to-source translation is Cetus[12]. Cetus does not provide support for Fortran, and is designed to facilitate parallelizing Java, C++, and C. Cetus is a successor in some ways to the Polaris compiler [13] which operated instead on Fortran 77. Also, various other source-to-source converters are not of concern to the work of this thesis because they deal with converting between languages, not modifying code in the manner proposed in this thesis. For example, there are UPC to C, CAF to F90, Java to C++, and countless other esoteric variants like SML to Java, or Lambda Calculus to Javascript.

Chapter 2

Source-To-Source Translation

Source-to-source translation is a process by which a source code is modified by a special purpose application and the resulting source code can then be used as desired. Source-to-source translation therefore can be platform independent, and can facilitate a number of interesting changes to a set of source code files. Some change for example can be applied to hundreds or thousands of files at once in a manner much faster than could be done by hand. The input and output languages from a translator may or may not be the same. This thesis only considers translation from C to C or C++ to C++. Some possible example changes include:

- Changing all variable names to be consistently formatted
- Translating a fortran program to a functionally equivalent C program
- Restructuring loops to facilitate optimizations(unrolling, jamming, merging)
- Inlining of functions
- Changing all function names in a library's source to be prepended with the library's name

Source-to-source translation should be done by an application that fully understands and can correctly parse the input language. A simple search-and-replace function of a text editor will not be able to correctly analyze the structure and symantics of most languages. For example, a regex based search will not be able to differentiate between a function and variable with the same name.

The general method for source-to-source translation is to parse the input source code into an abstract syntax tree(AST), then to manipulate the AST, and finally to generate source code from the modified AST. Figure 2.1 shows this flow from input to output, with an example of an actual AST produced by the PUP translator described in Chapter 5.

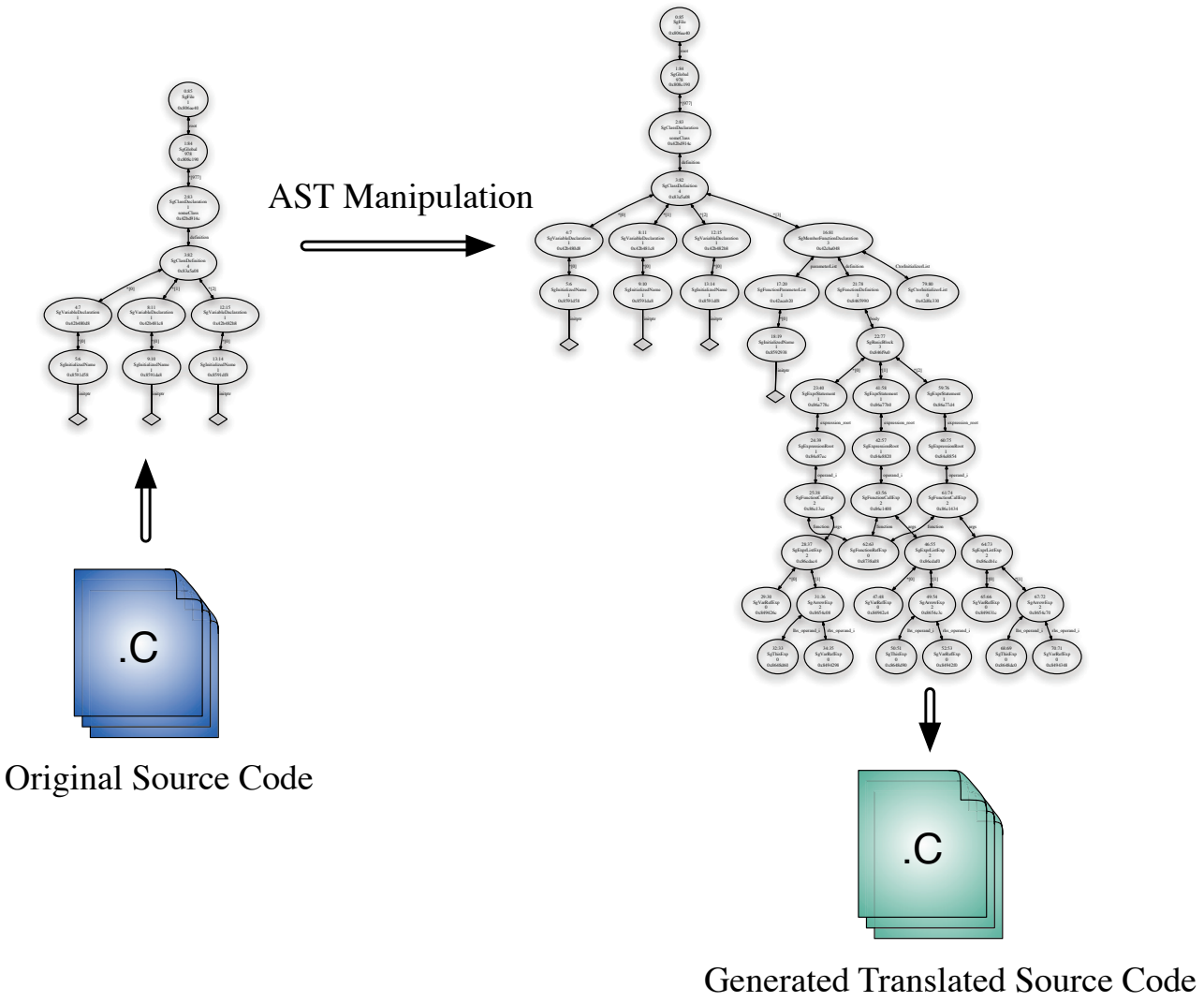

Figure 2.1: Source-to-source overview

2.1 ROSE for Source-to-Source Translation

ROSE is a library that can be used to write source-to-source translators[1, 14, 16, 15]. A ROSE based translator can parse, modify and output C or C++ code. The ROSE library is developed at Lawrence Livermore National Laboratories by Dan Quinlan and others. It currently supports C and C++ languages, although it has been used by some other group to support FORTRAN via a front-end parser called OPEN64. ROSE is not available on the internet, but may someday be freely released. Academic users might be able to obtain a copy from the ROSE team.

ROSE uses the EDG front-end for its parsing of C++ and C into the abstract syntax tree that is exposed to the user of the ROSE library. Each node in the tree will have a scope, a file info, some type of specifier for how the associated code should be generated after modification, a link to a parent, and various specific attributes such as a name. Some of the 241 types of AST nodes used by ROSE are listed in figure 2.2. The programmer reference documentation distributed with ROSE should be used to explore the various types of nodes. Each node has its own kinds of attributes and accessor functions. The details for each node is well beyond the scope of this thesis.

In addition to parsing source files into an AST, the ROSE library provides query functions that traverse the AST building a list of all nodes of the desired type. For example, it is easy to obtain a list of all variable declarations in a file or given scope. Iterators can easily scan through all variable reference expressions, or any other type of AST node. Various mechanisms are in place and are being developed in ROSE to allow for modification of the AST. Lower level rewrite mechanisms allow for modification, removal, and insertion of AST nodes while higher level mechanisms allow for insertion of arbitrary C++ code represented by a string to the AST in a specified location.

- `SgProject` A ROSE project that can contain multiple files
- `SgFile` A source-code file
- `SgFunctionDeclaration` A function declaration
- `SgFunctionDefinition` A function definition
- `SgFunctionCallExpr` A function call expression
- `SgFunctionRefExp` A function reference, to be used in a function call expression
- `SgBasicBlock` A block of code, surrounded by { and }
- `SgVarRefExp` A variable reference expression
- `SgVariableDeclaration` A variable declaration
- `SgClassDeclaration` A class declaration, either a forward declaration or a defining declaration
- `SgClassDefinition` A class definition
- `SgMemberFunctionDeclaration` A member function in a class
- `SgCtorInitializerList` for initialization lists at constructors
- `SgArrowExp` The "->" operator for pointers
- `SgThisExp` The pointer to the "this" object
- `SgWhileStmt` A while statement
- `SgMinusOp` The unary operator minus
- `SgPointerDerefExp` A pointer dereference expression
- `SgDeleteExp` The C++ delete operator

Figure 2.2: A selection of the 241 AST nodes used by ROSE

2.2 The Simplest Possible ROSE Translator

The simplest of all ROSE translators is the identity translator, an example of which is shown in figure 2.3. It takes the source for a program, parses it in to the internal AST format, and generates output code which should be almost identical to the initial input files. This translator is useful for debugging ROSE, and for initially testing new codes. For example the identity translator should always be used first when a new code is being translated. This will ensure the translator is being given correct compiler options, include directories, and ROSE language flags.

```
#include "rose.h"
int main( int argc, char * argv[] )
   {
          // Build the AST used by ROSE
          SgProject* project = frontend(argc,argv);

          // Insert your own manipulation of the AST here...

          generateDOT(*project); // Generate a graph for the AST
          generatePDF(*project); // Generate a pdf describing the AST

          // Generate source code from AST and call the normal compiler
          return backend(project);

   }
```

Figure 2.3: A simple identity translator that uses ROSE

All transformations are done through accesses to the `project` variable which contains pointers to the `SgFile` nodes in the project. Each file contains a scope node which can be used to further access the tree. Additionally queries can be performed directly on `project` or any other node to return a list of any desired type of node in the AST.

2.3 Complications

When using ROSE for the first time, a number of problems can arise, and the ROSE documentation is large, but has sparse coverage of the entire ROSE library and its use. A new user of ROSE should be aware of two considerable problems when developing translators. The first is an issue regarding language differences between C and C++ while the second is a simpler problem with include flags.

2.3.1 Languages

There are a number of subtle issues regarding dialects of C and C++. All C programs are not valid C++ programs. Thus in some cases the user of a translator must pass special flags to the ROSE-based translator to specify which language is used for a program.

This problem exists when parsing the NAS Parallel Benchmark IS. The NAS NPB benchmarks use a variable called `class` to store the desired benchmark problem size. In C++, `class` is a reserved keyword, which can therefore not be used as a variable name.

To parse such a C file, it may be necessary to use the command line option `-rose:C_only` or `-rose:C99_only`. Running the identity translator on a code should allow the user to quickly determine which flag should be used.

2.3.2 Compiler Flags

ROSE currently does not include "." as an include directory, so a flag such as "-I." should commonly be used. Additionally, a space is not allowed between the "I" and the following directory. Some common compilers allows such a space, but ROSE does not.

2.4 Stability and Robustness

Rose provides multiple different mechanisms for manipulating AST's, from a very low level tedious manual modification of the nodes of the tree and all their associated fields to a higher level method whereby a string of C++ code can be inserted at a particular location in the AST. The higher level methods are not robust or fully implemented, and thus are not used for the three translators described in this thesis. The low level method is seeming quite robust. Unfortunately learning how to use each particular type of AST node is non-intuitive at first, and much guessing and checking is required before learning what must be done in each case. There is no description of what is required for each node in a well-formed ROSE AST.

Currently there are only two known C codes which we have not yet managed to parse with even the identity translator. The first is a C program which includes the file `charm++.h` and the second is part of the Zoltan partitioning library. These may prove to be simple hurdles to overcome, or maybe ROSE has bugs which will be difficult to pinpoint.

Chapter 3

Thread Variable Privatization

3.1 Introduction

Amongst the various possible uses of source-to-source translation in parallel programming is the modification to remove global variables from C or C++ programs. Various user-level thread packages will require global variables to be handled differently when virtualization is applied. The term "virtualization" refers in this thesis to the use of multiple threads or virtual processors within a single process on one physical processor. Parallel frameworks such as AMPI[6, 3, 11, 7, 5] and various multi-paradigm languages currently under development will be able to benefit from this work.

In order to build a user-level thread based system, the thread-private state for each thread must be encapsulated and accessible by the currently running thread upon a thread context-switch. Often time the thread-private data is the stack, and associated internal processor state including the contents of various registers.

It is not always possible to just invent a new language and compiler for use in existing production applications. Because a huge amount of code for existing applications is already written in C, C++, and FORTRAN; it is best to try to utilize these languages as much as possible. Section 3.2 describes the layout of C and C++ programs in memory and why we must specifically address global and static variables. Section 3.3 addresses the issues with global and static variables in the context of virtualized MPI applications. The information provided in this chapter may also prove useful in contexts other than MPI, such as a virtualized version of a GAS language.

3.2 Data Layout for Variables in C and C++

Almost all compiled languages on traditional computers follow the same pattern for data layout. Source code is compiled into executable programs that use a single 32 or 64-bit virtual address space. The program, when executing, will be loaded into its virtual memory space in a number of segments. Figure 3.1 shows a simple view of the partitioning or segmenting of a program's address space. The exact layout depends upon the operating system, compilers, linkers and loaders used to compile and run the program. The typical method is to have a code segment, a data segment, a stack, and a heap. The code segment may be write-protected while the data segment is able to be written. The series of instructions to be executed will be loaded at runtime into the code segment. The data segment will contain constants and data compiled into the program. The heap is used at runtime from which memory is dynamically allocated. The stack will grow from one end of its segment as functions are called and their return pointers and variables are pushed onto the stack. In reality on many systems the layout can be more complicated, with more segments, complicated permission schemes, and varying names or designations for the regions of memory, but these further complications are beyond the scope of this thesis.

3.2.1 Globals and Statics: An Overview

Global variables are both simple and complicated. In a C program, it is trivial to create a global variable, however understanding how a global variable is handled by compilers, linkers, and loaders is more difficult. This section provides simple examples to show how global variables are compiled on current systems. Additionally this section provides example programs containing static variables, which are similar but not identical to globals. Table 3.1 lists the code from two files, and shows the relevant symbols from an executable compiled from these two source files. The symbols are produced by running the tool `nm` on the resulting executable. The program has 2 globals in the first file, one called `a`, and a static global called

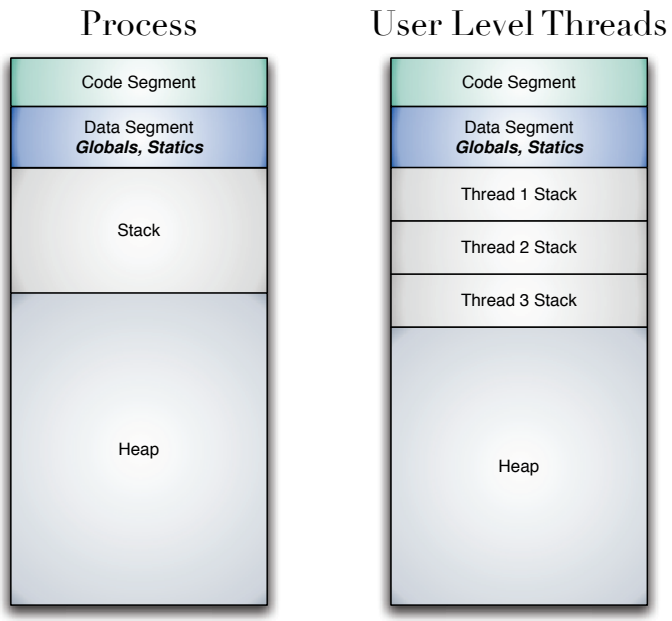

Figure 3.1: Memory layout for a program and a user-level threaded program

`b`. The second file utilizes the same global `a` as well as its own static global `b`. As expected in the symbols found in the executable are found two `b` variables, one for each file, and one `a` shared by both files. Additionally each file contains a function which in turn contains a static variable. These statics are not globals, but a similar effect happens when they are compiled, that is they are located along with the globals in a single memory segment. The symbols in the executable show that indeed two variables named `c.0` appear. These are for the two different static integers named `c`.

Source File 1	Source File 2	Symbols in Executable

int a;
static int b;
void f(void);
int main(){
  static int c;
  a=100;
  b=100;
  f();
}
	#include <stdio.h>
	int a;
	static int b;
	void f(void){
	  static int c;
	  printf("a=%d b=%d", a, b);
	  a=100;
	  b=0;
	}
		`08049614 B a`
		`08049620 b b`
		`08049628 b b`
		...
		`0804961c b c.0`
		`08049624 b c.0`
		...
		`080483b8 T f`
		`0804838c T main`

Table 3.1: 2 source-code files with global and static variables.

Global variables are accessible from any file in which they are declared, either with or without the use of the term `extern`. Global variables are all grouped together into a particular segment of memory when a program is run. Thus all files can know exactly how to reference the global. Static variables are variables whose values persist beyond exiting the scope of the variable. For this reason, a static variable is essentially a global variable with a limited scope. The static variable can only be accessed in its scope using the standard scoping rules. Thus a static variable can be implemented in the same manner as a global, with a sufficiently mangled name to distinguish between multiple static variables with the same name, but different scopes.

3.3 Global Variables with Virtualization

In an MPI application, the user expects that if one of the processors sets the value of a global variable, then the value will still be the same the next time that the same processor reads the value. Commonly this may occur as in Figure 3.2. Obviously the programmer will expect the value of `my_rank` not to change after setting it. In the most common MPI implementations, each MPI processor is assigned to an OS process running on a physical processor. In this case each process has its own set of global variables. Thus things work as expected. Unfortunately, under virtualization, things start breaking down.

```
#include "mpi.h"
int my_rank;

int main(){
    MPI_Init(...);
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    ...
    printf("myrank=%d\n", my_rank);
}
```

Figure 3.2: MPI example program with a global variable called `my_rank`

If multiple threads run sharing a single copy of the global, they may interfere with each

other, and each may not see its latest updates to a global variable. Figure 3.2 shows a not so farfetched example where a global variable is used in an MPI program. Thus in our example the actual memory location holding the `my_rank` should be unique for each MPI process. In AMPI, many virtual processors may execute within a single OS process. Thus we must have a mechanism for providing separate sets of global variables for each virtual processor. Section 3.3.1 describes one mechanism for providing separate sets of globals to each virtual processor; just require that the programmer remove any global variables. Vacuously, then, there is a null set of globals for each process. A better mechanism for providing separate sets of variables it to modify the GOT as described in Section 3.3.2. Section 3.3.3 discusses briefly how a compiler could eliminate global variables, but tells of the downsides to this approach. The scheme we propose is to use a source-to-source translator which removes all globals and statics from a program without changing the behavior of the program. Section 3.3.4 describes the source-to-source translator written for this purpose.

3.3.1 Manually Rewrite Source Code

One way of solving this problem for AMPI applications is to not use any global variables. The globals will therefore live on the stack or heap, not in the single overlapping section for the DATA segment. This is easily accomplished for the example 3.2 as shown in 3.3. The global variable is just declared in the main function.

```
#include "mpi.h"

int main(){
    int my_rank;
    MPI_Init(...);
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    ...
}
```

Figure 3.3: Code from figure 3.2 manually rewritten to eliminate the global variable

3.3.2 Swap Globals

Another way of solving this problem for AMPI applications is to have AMPI swap the globals in and out at each thread context switch. This solution requires a platform where it is possible to identify the memory locations for globals making it possible to replace the global data. On some platforms all global variables are removed from their actual data by a second layer of indirection. To access a global variable, a pointer to the data is stored in a table. On such platforms, AMPI can just store different copies of the table and modify the pointers in the table at a context switch.

A platform which supports swapping global variables by use of a table of pointers to the global variables is ELF. ELF uses a Global Offset Table(GOT), which contains a list of all global variables, and pointers to the actual data, along with other data describing the variables. The entries in this table can be modified easily, but the modifications will take small amounts of time. Unfortunately, the methods for accessing the ELF GOT varies across operating systems and architectures. Currently AMPI's swap-globals method supports 32-bit x86 linux. Extending it to support 64-bit ELF could probably be done as well.

A common platform which does not provide ELF is Apple's Mac OSX, which uses Mach-O executable format. All globals are accessed by a pointer computed as an offset from the contents of a register which is essentially the program counter. Thus globals can be more quickly accessed on Mach-O systems, but they lack the flexibility provided in ELF by its use of an extra level of indirection.

3.3.3 Compiler Based Solutions

Compilers could be modified to force global variables to be located on the stack inside of the main function. Such a solution would work with AMPI, where each thread has its own stack, and would hence have its own set of global variables. Unfortunately, modifying a compiler will limit the potential deployment platforms because users may not want to build or install

another set of compilers. Also, common compilers such as gcc are notoriously difficult to modify due to a lack of documentation for the compiler's codebase. Users will also have their preferences for compilers which are fast or work well with their applications, and many of those are proprietary, and thus not easily modified.

3.3.4 Automatically Rewrite Source Code

A source-to-source translator could be used to automatically encapsulate all global variables and put them on the stack. The approach of creating a source-to-source translator to encapsulate the global variables has proven effective so far. The translator we propose uses the ROSE library to parse and modify the source code for an application. The source for the translator is provided in Appendix A. The steps in the process are listed below. Dan Quinlan, the lead developer for ROSE, wrote an initial translator which which identified and moved global variables. It only worked in the simplest cases, it didn't handle statics, and it didn't handle multiple files well, but it led to our proposed translator.

- Build list of all global variables by iterating through entire AST
- Build list of static variables by iterating through entire AST
- Set output filenames to overwrite existing if desired
- Find file containing the main function: `main()`, `AMPI_Main()`or `AMPI_Main_cpp()`
- Create a struct(initially empty) to encapsulate all the globals and statics
- Create a new initializer function(initially empty)
- Build list of all variable references to statics or globals by iterating through entire AST
- Move all global variables into the class, removing their old declarations

- Move all static variables into the class, removing their old declarations, and mangling their names

- Declare an instance of the struct in the top of the main function

- Add a call to the initializer function in the main function, using the new instance of the struct

- Append a reference to the struct to all function definitions in the project

- Append the reference to the struct to all function call expressions

- Fixup all references to the globals or statics to be indirected through the parameter provided in all functions

- Remove all initializers from the globals and statics contained in the struct, and place equivalent statements in the initializer function

- Copy the struct definition from its one location in the file containing the main function to the top of all other files

- Remove extern declarations for any globals moved into the struct

The translator which was written following the structure listed above works with a great many samples. Unfortunately there is at least one known cases where this approach fails. Section 3.4 describes one of these problems and a proposed solution in the context of a real application we want to use with the translator.

3.3.5 Example translations

This section provides two example programs translated by a source-to-source translator which eliminates global and static variables. The first example shows a program with three global variables. The second example shows a program with multiple static variables, each

with the same name. The examples are meant to provide easy to read examples, not to show all the features of the translator .

Figure 6.6 lists the original code for a simple program which contains three global variables. Figure 6.7 lists the code output by the translator. The output code contains no globals, but does contain a struct which encapsulates the globals. The struct is instantiated in the top of main, and is added on as a pointer parameter to all other functions, in this case only `f()`. Tables 6.8 and 6.9 verify that the globals are present before translation, and they are eliminated by the translation.

The second example, which shows how statics are handled, is shown in Figures 6.10 and 6.11. The two statics have their names mangled to reflect their different scopes. To do this, the translator specifically requests a mangled name for each static variable from ROSE. Thus a complicated mangled name replaces the original name for the variable. Although seemingly complicated, a mangled name must be used if more than one static variable has the same name. The intializers are handled correctly in this case as well. For the example, if `main()` or `f()` had contained other statements, those would have remained, but this example is intended to be simple.

3.4 Case Study: MILC

This section presents a large MPI code named MILC. It describes the limited success so far with the translator and the plans for expanding the translator to work well with the complicated cases that arise when using this large real world application.

3.4.1 MIMD Lattice Computations

MILC is a code used to simulate and study quantum chromodynamics(QCD), a theory describing the strong interactions between subatomic particles. MILC, which stands for MIMD Lattice Computation, is a widely used large code with over a hundred files in its

distribution[4]. It is written in C, C++ and assembly, using MPI for its parallel version.

The benchmarking version of MILC has been analyzed to determine the extent of its usage of global variables. The object files produced when compiling MILC can be viewed with the `nm` utility to determine exactly how many global and static variables MILC contains. The object files contained 78 global variables which need to be dealt with before this code can work with AMPI. Figure 3.4 lists the global variables and the corresponding object files in which they were found. The globals are spread across 8 different files. Manually modifying MILC is therefore a complicated task. Thus having a source-to-source translation tool for automatically removing these variables is of great importance.

The translator described in this chapter does not yet work with this complicated example, but work is proceeding on making it translate the needed MILC files. The reason it does not yet work is that there are global variables with user defined types that are only within the scope of some of the files, namely those where the global is declared. Thus adding the variable to the stack allocated struct is non-trivial. There have been various discussed methods for handling the user defined types, and the best solution proposed so far will require many additional transformations of the code. First we will create a `getsize_name()` function for each global variable in the file where such a global variable exists. This function will return the size of the global variable by simply calling `sizeof(global_variable`. Additionally a non-defining function declaration for each `getsize_name()` function, i.e. a prototype, must be added to the file containing `main()`. Then the struct will contain a pointer instead of an actual declaration of the variable. Then in `main()` each variable will be allocated using `alloca(getsize_name())`. Finally, when rewriting all references to the globals, the variable will have to be dereferenced from the struct since it is now a pointer. This solution should work for all possible examples we could foresee, including `typedef` and `class` global variables. The situation is similarly complicated for static variables of a user defined type.

gaugefix2.o:
- diffmat_offset
- sumvec_offset

setup.o:
- gf
- par_buf

gauge_stuff.o:
- gauge_action_description
- loop_char
- loop_coeff
- loop_expect
- loop_ind
- loop_length
- loop_num
- loop_table

control.o:
- beta
- ensemble_id
- epsilon
- even_sites_on_node
- fpi_mass
- fpi_nmasses
- gen_pt
- g_ssplaq
- g_stplaq
- iseed
- lattice
- linktrsum
- mass1
- mass2
- nflavors1
- nflavors2
- niter
- node_prn
- npbp_reps_in
- n_sources
- nt
- number_of_nodes
- nx
- ny
- nz
- odd_sites_on_node
- phases_in
- propinterval

control.o:
- rsqmin
- rsqprop
- savefile
- saveflag
- sequence_number
- sites_on_node
- source_inc
- source_start
- spectrum_multimom_low_mass
- spectrum_multimom_mass_step
- spectrum_multimom_nmasses
- spectrum_request
- startfile
- startflag
- start_lat_hdr
- startlat_p
- steps
- stringLFN
- t_fatlink
- this_node
- t_longlink
- total_iters
- trajecs
- u0
- valid_fatlinks
- valid_longlinks
- volume
- warms

reunitarize2.o:
- av_deviation
- max_deviation

quark_stuff.o:
- num_basic_paths
- path_num

d_congrad5_fn.o:
- cg_p
- resid
- t_dest
- ttt

fermion_force_asqtad3.o:
- backwardlink
- tempmom

Figure 3.4: Global Variables in compiled MILC object files

3.5 Case Study: WRF

This section presents a large MPI code named WRF. We first give a brief description of WRF and then describe why the global variables in this example cannot be eliminated by my translator. Finally we discuss the future plans for extending my translator to work with FORTRAN codes, and hence the important interesting code WRF.

3.5.1 Description of WRF

WRF is a Weather Research and Forcasting modeling system, developed by the National Center for Atmospheric Research(NCAR)[17, 2]. It is an advanced 3-D fluid dynamics code that incorporates a variety of models for air, land, and oceans. Its goal is "to provide a next-generation mesoscale forecast model and data assimilation system that will advance both the understanding and prediction of mesoscale weather and accelerate the transfer of research advances into operations."

WRF uses multiple levels of nested structured grids with microphysics, cumulus parameterizations, surface physics, planetary boundary layer physics, and atmospheric radiation physics.

3.5.2 Globals and Statics Variables in WRF

I built a copy of WRF to determine what global variables exist in its executables. My plan was then to determine if my translator would be able to eliminate these global variables. Unfortunately WRF contains a number of FORTRAN files which currently cannot be handled by my translator. Because many scientific and engineering applications use FORTRAN, the translator must be extended to support this currently unsupported language. Adding fortran support is currently a major desire for the PPL group members working with AMPI applications.

WRF can be built with the command "compile em_b_wave". This produced two im-

Executable	Number of Globals
ideal.exe	43
wrf.exe	61

Figure 3.5: Count of Global BSS symbols in WRF executables

portant executables on the NCSA IBM AIX machine named copper. The first executable is ideal.exe. The second is wrf.exe. The executables were examined with the "nm" tool and a count of the Global BSS symbols is reported in table 3.5. These variables are from FORTRAN modules, which are similar to C global variables. `wrf.exe` contains 61 globals while `ideal.exe` contains 43.

&&N&@data_info	mpi_status
&&N&@esmf_calendarmod	old_offsets
&&N&@esmf_timemod	point_move_receives
&&N&@module_configure	point_move_sends
&&N&@module_date_time	pr_descriptors
&&N&@module_dm	regular_decomp
&&N&@module_domain	rslMPIHandleLUT
&&N&@module_ext_internal	rsl_debug_flg
&&N&@module_io	rsl_mpi_communicator
&&N&@module_machine	rsl_myproc
&&N&@module_nesting	rsl_ndomains
&&N&@module_quilt_outbuf_ops	rsl_noprobe
&&N&@module_wrf_error	rsl_nproc
&&N&@module_wrf_quilt	rsl_nproc_all
&&N&@wrf_data	rsl_nproc_m
__C_runtime_pstartup	rsl_nproc_n
cdf_routine_name	rsl_padarea
domain_info	rslsysxx
io_seq_compute	sh_descriptors
io_seq_monitor	sw_allow_dynpad
mess	xp_descriptors
mh_descriptors	

Figure 3.6: 43 Global BSS symbols in WRF executable ideal.exe

&&N&@data_info	__C_runtime_pstartup
&&N&@esmf_calendarmod	cdf_routine_name
&&N&@esmf_timemod	domain_info
&&N&@module_configure	idx_
&&N&@module_date_time	input
&&N&@module_dm	io_seq_compute
&&N&@module_domain	io_seq_monitor
&&N&@module_ext_internal	mess
&&N&@module_io	mh_descriptors
&&N&@module_machine	mpi_status
&&N&@module_mp_etanew	old_offsets
&&N&@module_mp_ncloud3	point_move_receives
&&N&@module_mp_ncloud5	point_move_sends
&&N&@module_mp_thompson	pr_descriptors
&&N&@module_mp_wsm3	regular_decomp
&&N&@module_mp_wsm5	rslMPIHandleLUT
&&N&@module_mp_wsm6	rsl_debug_flg
&&N&@module_nesting	rsl_mpi_communicator
&&N&@module_progtm	rsl_myproc
&&N&@module_quilt_outbuf_ops	rsl_ndomains
&&N&@module_ra_gfdleta	rsl_noprobe
&&N&@module_ra_gsfcsw	rsl_nproc
&&N&@module_ra_rrtm	rsl_nproc_all
&&N&@module_sf_myjsfc	rsl_nproc_m
&&N&@module_sf_noahlsm	rsl_nproc_n
&&N&@module_sf_sfclay	rsl_padarea
&&N&@module_wrf_error	rslsysxx
&&N&@module_wrf_quilt	sh_descriptors
&&N&@module_wrf_top	sw_allow_dynpad
&&N&@wrf_data	xp_descriptors
&&N&module_ra_gfdleta	

Figure 3.7: 61 Global BSS symbols in WRF executable wrf.exe

Chapter 4

Automated Performance Tracing

It is critical to be able to accurately analyze the performance of large scale high-performance applications. Various languages and parallel systems provide different tools for analyzing performance. Charm++ and AMPI support tracing and analyzing performance with the tool named Projections. Projections contains many tools for analyzing time varying data including processor utilization, message statistics, timelines, and a number of histograms and other views.

When analyzing performance in both serial and MPI applications, an important metric is the amount of time spent in each function. Currently the Projections performance analysis tool does not rewrite binaries, so it relies upon tracing calls made by the runtime communication library as well as user written tracing calls. Thus to get the most detailed performance data in an AMPI application, the user must explicitly write some special function calls. These calls inform the tracing library when a particular function or piece of code starts and ends. The calls are simple, and can therefore be easily inserted automatically. This chapter describes and shows an example of the source-to-source translator for automatically generating and inserting tracing calls.

4.1 AMPI Performance Tracing with Projections

AMPI supports a set of function calls for tracing nested functions. These functions have prototypes and simpler macro wrappers in `ampi.h` which is identical to the file called `mpi.h` provided by AMPI.

- `traceBeginFuncProj()` specifies the beginning of a function

- `traceEndFuncProj()` specifies the end of a function

- `traceRegisterFunction()` called once at the beginning of the program to register each function which is traced

Often a program will have a stack of function calls. If all these functions are traced, the analysis is complicated by the excessive data. For example, a multi-physics code may just want to trace the time spent in each of its three numerical solvers, and therefore the calls to each solver are bracketed with the functions above.

4.2 Performance Tracing Source-to-Source Translator

We implemented a translator which automatically inserts the calls listed above into an application's source code. The translator first identifies functions within a specified call depth from `main()`. It then inserts tracing calls inside each of the definitions of each of these functions. `traceBeginFuncProj()` is inserted at the beginning of each function, and `traceEndFuncProj()` is called at the end of each function and also preceding all `return` statements in the function. Each function that is traced also has a corresponding `traceRegisterFunction()` added to `main()`.

The call depth is a configurable command line argument to the translator. This allows a user to control the degree of detail that will appear in the resulting traced performance data. When planning this translator, we decided we wanted a simple mechanism for specifying how much of an application to trace. In the future, if it becomes helpful, we will add some means for a user to specify which files to not trace or files to add to those traced.

The Projections runtime and post-mortem tools will compute the exclusive times spent in each of the traced functions. Thus applicable tools can be written to display various types of performance analyses. Currently we are developing tools inside Projections for AMPI

now that we have a means for automatically tracing large programs. This project is active work with new graphical display tools currently being written, so performance results from applications are not described here.

4.3 Example

Figure 6.3 shows an example program that has a `main()` function and three other functions, `A()`, `B()`, and `C()`. The functions are called one from another, starting with main. The purpose of having this chain of functions is to illustrate that this source-to-source translator is capable of tracing nested functions to a user selected depth. Figure 6.4 shows the resulting translated code after tracing calls were inserted to a depth of 3. Figure 6.5 shows the resulting translated code when the depth of tracing was specified to be 2. The main difference is that function `C()` is not traced in the later case because it is at a call depth of 3 which is greater than the specified 2. The simplicity of specifying just a simple integer depth is quite useful, but yet also provides a great deal of control over the amount of tracing data recorded.

Chapter 5

Automatic PUP Function Creation

5.1 PUP Functions

Charm++ is a parallel runtime system and programming model which provides support for migratable objects composed of user written C++ classes along and a simple interface definition. Charm++ does not use its own compiler because it is not a language itself except for its simple translation of the short interface definitions into C++. The decision not to create and use a special purpose compiler has limited the available code analysis for help with various useful parallel functions such as the serialization of migratable objects. Charm++ currently requires uses to write Pack-and-UnPack (PUP) functions for each migratable object. See Figure 5.1 for an example of a PUP function. Each PUP function has a single pupper as its parameter, often named `p`, which has an associated overloaded vertical bar operator. The pupper has internal state specifiying whether it is in a packing state or an unpacking state, and this internal state controls the behavior of the overloaded vertical bar. Thus only a single function is required in order to perform both the packing and unpacking functions which serialize and unserialize an object into or from some buffer.

PUP functions are used not only for migrating objects, but is also used for checkpointing and restarting objects to and from disk or remote memories, serializing some graphics data when using LiveViz and CCS, and other miscellaneous tasks.

Users are currently required to write their own PUP functions, but a compiler based solution or an automated source-to-source solution would eliminate the need for users to be concerned with writing PUP functions and keeping them up to date whenever a class is

```
#include <pup.h>
class exampleClass
{
 int a;
 double b;
 char c;

 void PUP(class PUP::er &p)
  {
    p|a;
    p|b;
    p|c;
  }
};
```

Figure 5.1: A C++ class with a PUP function

modified. Generally writing PUPs is considered simple, but a hassle.

5.2 PUP Creator Source-to-Source translator

This thesis presents a simple PUP creation source-to-source translator. It takes in a set of C++ source files and will add to each class a PUP function. The PUP function will be populated with statements for PUPing each of the member variables in the class. Currently the translator generates a call in the form `p|a;` for each class member variable. This works for many cases, although arrays and pointers should be handled differentlly. This translator could be extended to handle arrays and pointers. The main trouble with this extension is that it may not be possible to determine the size of an array at compile time. One downside to automatically creating PUP functions is that a user knows the minimal set of state which should be PUPed. It is non-trivial for a compiler to determine this set of data. For example, it may be faster to compute a set of values than transmit them over a network, or maybe the programmer can easily determine which variables are live at the point where they are PUPed. This leads to one disadvantage of the current implementation; all member variables are PUPed. In an ideal version, a minimal amount of data should be PUPed since the

serialized version will be written to disk, stored in memory, or sent over a network. Live variable analysis is not currently implemented in ROSE, but it could potentially be added if needed. A few other concerns which are yet unaddressed are whether to PUP inherited members. Currently we do not PUP any inherited members.

5.3 Example

Figure 6.1 and 6.2 shows a C++ source file before and after the PUP calls are added by my source-to-source automatic PUP creation tool. One curious result is that the explicit use of the `(this)->` code generated by the translator. Note that `(this)-> a` is equivalent to `a` inside a class in most cases. The obvious exception is when another variable "a" is in a local scope and the class variable is hidden. The files displayed in the figure are the exact code consumed and produced by my translator.

Chapter 6

Example Translated Codes

This chapter contains examples of translated codes for each of the source-to-source translators described in the thesis. Chapters 3,5, and 4 describe these examples and their motivations.

```
#include <pup.h>

class someClass {
public:
    int a;
    double b;

private:
    char c;

};
```

Figure 6.1: A C++ file before the automatic creation of PUP functions

```
#include <pup.h>

class someClass
{
  public: int a;
  double b;
  private: char c;


  public: void PUP(class PUP::er &p)
 {
    p|(this) -> a;
    p|(this) -> b;
    p|(this) -> c;
  }

}

;
```

Figure 6.2: A C++ file after the automatic creation of PUP functions

```
#include <mpi.h>

int C(){
  if(0==1)
        return 0;
  else
        return 1;

}

void B(){
  C();
}


int A(int j){
  B();
  return 7;
}


int main(int argc, char** argv){
        A(1);
}
```

Figure 6.3: A sample program before tracing calls are inserted

```
#include <mpi.h>

int C()
{
  traceBeginFuncProj("C","UnknownFile",1);
  if ((0 == 1)) {
    traceEndFuncProj("C");
    return 0;
  }
  else {
    traceEndFuncProj("C");
    return 1;
  }
  traceEndFuncProj("C");
}


void B()
{
  traceBeginFuncProj("B","UnknownFile",1);
  C();
  traceEndFuncProj("B");
}


int A(int j)
{
  traceBeginFuncProj("A","UnknownFile",1);
  B();
  traceEndFuncProj("A");
  return 7;
  traceEndFuncProj("A");
}


int main(int argc,char **argv)
{
  traceRegisterFunction("main",-999);
  traceBeginFuncProj("main","UnknownFile",1);
  traceRegisterFunction("A",-999);
  traceRegisterFunction("B",-999);
  traceRegisterFunction("C",-999);
  A(1);
  traceEndFuncProj("main");
  traceEndFuncProj("main");
}
```

Figure 6.4: Program of Figure 6.3 after tracing calls are inserted to a depth of 3.

```
#include <mpi.h>

int C()
{
  if ((0 == 1)) {
    return 0;
  }
  else {
    return 1;
  }
}


void B()
{
  traceBeginFuncProj("B","UnknownFile",1);
  C();
  traceEndFuncProj("B");
}


int A(int j)
{
  traceBeginFuncProj("A","UnknownFile",1);
  B();
  traceEndFuncProj("A");
  return 7;
  traceEndFuncProj("A");
}


int main(int argc,char **argv)
{
  traceRegisterFunction("main",-999);
  traceBeginFuncProj("main","UnknownFile",1);
  traceRegisterFunction("A",-999);
  traceRegisterFunction("B",-999);
  A(1);
  traceEndFuncProj("main");
}
```

Figure 6.5: After inserting tracing calls to a depth of 2

```
int global_a;
float global_b;
unsigned global_c;

int f(int j){ return j + global_c; }

int main(int argc, char** argv){
    f(global_a);
}
```

Figure 6.6: An example program with multiple global variables

```
struct AMPI_globals_t
{
  int global_a;
  float global_b;
  unsigned int global_c;
};

int f(int j,struct AMPI_globals_t *AMPI_globals)
{  return ((j) + AMPI_globals -> global_c); }

int main(int argc,char **argv)
{
  struct AMPI_globals_t AMPI_globals_on_stack;
  f((&AMPI_globals_on_stack) -> global_a,&AMPI_globals_on_stack);
}
```

Figure 6.7: Translated version of program from Figure 6.6. Contains no global variables.

Address	Type	Name
00000000	T	f
00000004	C	global_a
00000004	C	global_b
00000004	C	global_c
0000000d	T	main

Figure 6.8: Symbols created when compiling the code in Figure 6.6

Address	Type	Name
00000000	T	f
0000000e	T	main

Figure 6.9: Symbols created when compiling the code in Figure 6.7

```
void f(){
        static int a=3;
}


int main(int argc, char** argv){
    static int a=2;
}
```

Figure 6.10: An example program with two static variables. Both static variables are named a, but they have different scopes.

```
struct AMPI_TL_Vars
{
  int f___Fb_v_Gb__Fe___scope____SgSS2____scope__a;
  int main___Fb_i_Gb_i__sep____Pb____Pb__c__Pe____Pe\
___Fe___scope____SgSS2____scope__a;
}

;

void AMPI_TL_Vars_Init(struct AMPI_TL_Vars *AMPI_TLs)
{
  AMPI_TLs -> f___Fb_v_Gb__Fe___scope____SgSS2____scope__a = 3;
  AMPI_TLs -> main___Fb_i_Gb_i__sep____Pb____Pb__c__Pe____\
Pe___Fe___scope____SgSS2____scope__a = 2;
}


void f(struct AMPI_TL_Vars *AMPI_TLs)
{
}


int main(int argc,char **argv)
{
  struct AMPI_TL_Vars AMPI_globals_on_stack;
  AMPI_TL_Vars_Init(&AMPI_globals_on_stack);
}
```

Figure 6.11: After translation, the two static variables are replaced with two variables with mangled names are in the struct. The end of the mangled name is the original name a.

Chapter 7

Conclusions

Parallel programming can be difficult and time consuming. Using tools such as source-to-source translators can ease the burden of parallel programming. This thesis proposes a methodology and examples of source-to-source translation to simplify various aspects of parallel programming and performance analysis. The first translator eliminated global and static variables from a C or C++ application. The second translator inserted performance tracing calls into an MPI application for use with the Projections performance analysis tool. The final translator wote PUP functions for C++ classes. These translators were written using the ROSE library, and they can handle a wide range of C and C++ applications. FORTRAN is not yet supported by these translators, but FORTRAN support may be added in the future. The output resulting code from the translators is very similar to the input code except for the modifications performed by the translator. This thesis proposes that source-to-source translation is a good approach to dealing with various parallel programming tasks.

Appendix A

Thread Variable Privatization: Code Listing

The following is the source code for the Thread Variable Privatization tool described in Chapter 3. The main source file for this translator is `globalVariableRewrite.C`. Additionally the shared code described in Appendix D is required. The latest versions are available in the PPL group's CVS repository under the `ROSE-Translators` module.

```
/*
Adapted by Isaac Dooley from the file CharmSupport.C written by the ROSE group at LLNL.

Purpose: Encapsulate all Thread Local(TL) variables in a structure declared on
the stack, and pass around a pointer to this structure.

TL variables include global and statics

Why:

AMPI can only handle global variables on certain platforms. Basically
each thread needs its own thread-local copy of the global variables,
but each thread runs in a single process which has only a single set
of global variables. On ELF based systems, we can modify the ELF
Global Offset Table on thread context switches, but on systems with
Mach based linkers/loaders, all code is position independent, and
globals are hard coded relateive to the PC, so on thread contexts,
there is no efficient way of swapping in and out global variables
short of swapping in and out the entire DATA Segment(which can be
done). Thus rewriting global variables with a compiler is the best
solution to this problem, as it will be efficient as well as platform
independent.

TODO:  remove the placeholder variable after everything is built.
            Only do modifications if some statics or globals are found

*/

#include <rose.h>
#include <list>
#include <vector>
#include <algorithm>
#include <iostream>
#include <exception>


#include "globalVariableCommon.h"


#define DEBUG 0

using namespace std;

/* The MiddleLevelRewrite mechanism is not yet robust enough to use.  */
/* Unfortunately the current low level rewrite doesn't handle include statements at the top correctly */
/* Unfortunately the Middle level rewrite doesn't handle multiple files correctly */

#define USE_MIDDLELEVELREWRITE 0

bool overwrite_existing_files;

const char *structName = "AMPI_TL_Vars";
const char *initFuncName = "AMPI_TL_Vars_Init";
const char *placeHolderName = "___AMPI_place_holder";
```

```
/** main files contain a function that looks like main */
bool isFileMain(SgFile *file){
    // scan through all function declarations and see if any of them are main()
    list<SgNode*> functions = NodeQuery::querySubTree (file,V_SgFunctionDeclaration);
    for (list<SgNode*>::iterator i = functions.begin(); i != functions.end(); i++) {
        if(isFunctionMain(isSgFunctionDeclaration(*i)))
            return true;
    }
    return false;
}

/** builtin functions should start with double underscore */
bool isFunctionBuiltin(SgFunctionDeclaration *func_decl){
    char *func_name = func_decl->get_name().str();
    if(strlen(func_name)<2)
        return false;
    else if(func_name[0] == '_' && func_name[1] == '_' )
        return true;
    else
        return false;
}


/** Lookup the correct struct to use for the references to the global variables */
SgExpression* lookupTLStruct(SgNode* whichNode){

  // walk up AST to the functionDeclaration containing this FunctionCallExp
  SgNode* n = whichNode;
  while( ! isSgFunctionDeclaration(n) )
        n = n->get_parent();
  SgFunctionDeclaration *parentFunctionDeclaration = isSgFunctionDeclaration(n);
  assert(parentFunctionDeclaration);

  // If we are not in main, use final parameter from parameter list
  if(! isFunctionMain(parentFunctionDeclaration)){
        SgFunctionParameterList *fpl = parentFunctionDeclaration->get_parameterList();
        assert(fpl);
        list<SgInitializedName*> &args = fpl->get_args(); // really a std::list<>
        assert(args.size() > 0);
        SgInitializedName *finalArg = args.back();
        assert(finalArg);
        assert(finalArg->get_scope());
        SgVariableSymbol *variableSymbol = new SgVariableSymbol(finalArg);
        assert(variableSymbol);

        Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
        assert(fileinfo != NULL);
        SgVarRefExp *varRefExp = new SgVarRefExp (fileinfo, variableSymbol);
        return varRefExp;
  }
  else { // we are in main use the struct we created

        // Get main's basic block (currently parentFunctionDeclaration is main)
        SgBasicBlock *basicBlock = isSgBasicBlock(parentFunctionDeclaration->get_definition()

        assert(basicBlock);

        // Find first statement in the basic block
        SgStatementPtrList &statements = basicBlock->get_statements();
        // TODO FIXME: assume the first statement in main is our struct declaration
        SgStatement *firstStatement= *(statements.begin());
        assert(firstStatement);
        SgVariableDeclaration *variableDeclaration = isSgVariableDeclaration(firstStatement);
        assert(variableDeclaration);

        // Get the first variable defined in variable Declaration
        // There can be multiple i.e. "int i,j,k;"
        SgInitializedNamePtrList &args = variableDeclaration->get_variables();
        assert(args.size() == 1);
         // assume that the front=only variable being declared is the one we want
        SgInitializedName *finalArg = args.front();
        assert(finalArg);
        //assert(finalArg->get_scope());
        SgVariableSymbol *variableSymbol = new SgVariableSymbol(finalArg);
        assert(variableSymbol);

        // Create a reference to the struct variable
        Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
        assert(fileinfo != NULL);
        SgVarRefExp *varRefExp = new SgVarRefExp (fileinfo, variableSymbol);

        // Create an address of the reference to the struct
        fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
        assert(fileinfo != NULL);
        SgAddressOfOp *pointer = new SgAddressOfOp(fileinfo, varRefExp,NULL);

        return pointer;
  }
  return NULL;
```

```
}


/** Reassociate Preprocessor statements and comments from a given node
    This is mostly copied from rewriteLowLevelInterface.C
*/
void reassociatePreprocessorDeclarations(SgLocatedNode* fromHere, SgLocatedNode* toHere){
  // Get attached preprocessing info
     AttachedPreprocessingInfoType *comments = fromHere->getAttachedPreprocessingInfo();

     if (comments != NULL){

#ifdef DEBUG
        printf ("Found attached comments (at %p of type: %s): \n",fromHere,fromHere->sage_class_name());
#endif

        if( toHere->getAttachedPreprocessingInfo() != NULL){
            toHere->getAttachedPreprocessingInfo()->merge(*comments);
        }
        else {
            toHere->set_attachedPreprocessingInfoPtr(comments);
        }

        for(std::list<PreprocessingInfo*>::iterator i=comments->begin(); i!=comments->end(); ++i){
        }

        fromHere->set_attachedPreprocessingInfoPtr(NULL);


    }

}


/** Cleanup struct by removing the temporaray place holder variable declaration */
void fixupClassDeclarationPlaceHolder(SgClassDeclaration *classDeclaration){

        list<SgNode*> nodeList = NodeQuery::querySubTree ( classDeclaration, V_SgVariableDeclaration );

        list<SgNode*>::iterator i;
        for(i=nodeList.begin(); i!=nodeList.end(); ++i) {
                SgVariableDeclaration *varDecl = isSgVariableDeclaration(*i);
                assert(varDecl != NULL);

                // if this variable is called "place_holder" then get rid of it
                list<SgInitializedName*> & variableList = varDecl->get_variables();
                list<SgInitializedName*>::iterator var;
                for(var=variableList.begin(); var != variableList.end(); ++var) {
                        if( (*var)->get_name().getString() == string(placeHolderName) ){
                                cout << "Removing " << placeHolderName << " variable" << endl;
                                SgClassDefinition *parent = isSgClassDefinition(varDecl->get_parent());
                                assert(parent!=NULL);
                                parent->get_members().remove(varDecl);
                        }
                }
        }

}



/** Move all initializers from the variables in the classDeclaration to */
/**  assignment statements in the init function */
void fixupInitializers(SgClassDeclaration *classDeclaration, SgFunctionDeclaration *TLInit){
    Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

    list<SgDeclarationStatement*> decls = classDeclaration->get_definition()->get_members();
    list<SgDeclarationStatement*>::iterator i;

    for(i=decls.begin(); i != decls.end(); ++i) {
            SgVariableDeclaration *vardecl;
            if(vardecl=isSgVariableDeclaration(*i) ){
                    list<SgInitializedName*> names = vardecl->get_variables();
                    for(list<SgInitializedName*>::iterator n=names.begin();n!=names.end();++n){
                            SgInitializer* initializer = (*n)->get_initializer();
                            if(initializer != NULL){
                                SgAssignInitializer *assignInit;
                                if(isSgConstructorInitializer(initializer))
                                    cout << "WARNING: Can't yet handle SgConstructorInitializers" << endl;

                                if(isSgAggregateInitializer(initializer))
                                    cout << "WARNING: Can't yet handle SgAggregateInitializer" << endl;

                                if((assignInit=isSgAssignInitializer(initializer))!=NULL){
                                    SgExpression* initexpr = assignInit->get_operand();



                                    // figure out what is the parameter(the struct) passed in
                                    SgExpression *TLStruct = lookupTLStruct(TLInit->get_definition());
                                    assert(TLStruct);
```

```
                                        SgType *type = new SgTypeInt();

                                        SgVariableSymbol * sym = new SgVariableSymbol(*n);
                                        SgVarRefExp *varRefExp = new SgVarRefExp(fileinfo, sym);
                                        // Dereference the struct to get the variable
                                        SgArrowExp* redirectedReference = new SgArrowExp(fileinfo,
                                             TLStruct,varRefExp, type);

                                        // Figure out exactly what we are assigning (rhs)
                                        SgIntVal *v = new SgIntVal(fileinfo, 7);

                                        // create assignment statement
                                        SgAssignOp * assignOp = new SgAssignOp(fileinfo, redirectedReference,
                                                                               initexpr, type);
                                        SgExpressionRoot *exprRoot = new SgExpressionRoot(fileinfo,
                                                                               assignOp, type);
                                        SgExprStatement* exprStatement = new SgExprStatement(fileinfo,
                                                                               exprRoot);


                                        // insert assign statement into the init function
                                        SgFunctionDefinition* funcDef = TLInit->get_definition();
                                        SgBasicBlock *bb = funcDef->get_body();
                                        bb->append_statement(exprStatement);

                                    }

                                    (*n)->set_initializer(NULL);
                                }

                            }
                        }
                    }

}


/** Add a variable declaration to main that encapsulates all the globals. */
void declareClassAndInitInMain(SgGlobal* globalScope, SgClassDeclaration *classDeclaration,
                               SgFunctionDeclaration *TLInit){
  assert(globalScope != NULL);

  // Find the definition of main()
  list<SgDeclarationStatement*>::iterator i = globalScope->get_declarations().begin();
  while(i != globalScope->get_declarations().end())
        {
          SgFunctionDeclaration *functionDeclaration = isSgFunctionDeclaration(*i);
          if (functionDeclaration != NULL){
                if(isFunctionMain(functionDeclaration)){

                  // We must check for the function definition,
                  // because we may be looking at a predeclaration/prototype for main
                  // At first this is not obvious that anyone would do this,
                  // but AMPI does it to rename main.

                  SgFunctionDefinition *fdef = isSgFunctionDefinition(functionDeclaration->
                                                                      get_definition());
                  if(fdef){
                        SgBasicBlock* block = isSgBasicBlock(fdef->get_body());
                        assert(block);

                        // Create a variable declaration
                        Sg_File_Info* fileinfo =
                                        Sg_File_Info::generateDefaultFileInfoForTransformationNode();
                        assert(fileinfo != NULL);
                        SgClassType* variableType =  new SgClassType(classDeclaration->
                                                get_firstNondefiningDeclaration());
                        assert(variableType != NULL);
                        SgName *varname = new SgName("AMPI_globals_on_stack");
                        SgVariableDeclaration* variableDeclaration =
                                        new SgVariableDeclaration(fileinfo,*varname,variableType);
                        varname->set_parent(variableDeclaration);
                        variableDeclaration->set_parent(block);
                        assert(variableDeclaration != NULL);


                        // Create a function call expression statement
                        SgFunctionSymbol *funcSymbol = new SgFunctionSymbol(TLInit);
                         // TODO: Probably not correct, but it works for now
                        SgFunctionType *funcType = new SgFunctionType(new SgTypeVoid());
                        SgFunctionRefExp* functionRefExp = new SgFunctionRefExp(fileinfo, funcSymbol,
                                                                               funcType);
                        SgExprListExp* exprlist = new SgExprListExp(fileinfo);
                        SgFunctionCallExp* funccall =
                                        new SgFunctionCallExp(fileinfo, functionRefExp, exprlist,
                                                              funcType);
                        SgExpressionRoot *exprRoot = new SgExpressionRoot( fileinfo, funccall,
                                                                               funcType);
                        SgExprStatement * exprStatement = new SgExprStatement(fileinfo, exprRoot);
```

```
                        // First prepend the init function call, then prepend the struct declaration
                        block->prepend_statement(exprStatement);
                        // Insert the variable declaration into the body of main()
                        block->insert_statement(block->get_statements().begin(),variableDeclaration);
                        variableDeclaration->set_parent(block);

                    }

                }
            }
            i++;
        }
}



/** Rewrite all function call and declaration parameter lists to include the encapsulated globals */
void addTLClassAsParameter(SgNode* subtree, SgClassDeclaration *classDeclaration){

    map<string,int,less<string> > modifiedFunctionNames;

    assert(subtree != NULL);
    // Add parameter to function declarations

    list<SgNode*> functionDecls = NodeQuery::querySubTree (subtree,V_SgFunctionDeclaration);
    cout << "Found " << functionDecls.size() << " FunctionDeclarations in this subtree" << endl;

    list<SgNode*>::iterator i;
    for (i = functionDecls.begin(); i != functionDecls.end(); ++i)  {

        SgFunctionDeclaration *functionDeclaration = isSgFunctionDeclaration(*i);
        assert(functionDeclaration != NULL);

        if (    ! isFunctionMain(functionDeclaration) &&
                ! isStatementInHeader(functionDeclaration) &&
                ! functionDeclaration->get_file_info()->isCompilerGenerated()
            ) {

                //create the parameter
                SgClassType* variableType = new SgClassType(classDeclaration->
                                    get_firstNondefiningDeclaration());
                assert(variableType != NULL);

                SgName var1_name = "AMPI_TLs";
                SgPointerType *ref_type = new SgPointerType(variableType);
                SgInitializer * var1_initializer = NULL;
                SgInitializedName *var1_init_name=new SgInitializedName(var1_name, ref_type,
                                                    var1_initializer, NULL);
                var1_init_name->set_scope(functionDeclaration->get_scope());
                assert(var1_init_name->get_scope());


                  // Insert argument in function parameter list
                  SgFunctionParameterList *parameterList = functionDeclaration->get_parameterList();
                  parameterList->append_arg(var1_init_name);
                  var1_init_name->set_parent(parameterList);
                  assert(var1_init_name->get_parent());
                  var1_init_name->set_scope(functionDeclaration->get_scope());

                  // Add to the list of modified functions
                  modifiedFunctionNames[functionDeclaration->get_name().str()] = 1;
                }
        }

    // Build a list of function calls within the AST
    list<SgNode*> functionCallList = NodeQuery::querySubTree (subtree,V_SgFunctionCallExp);
    cout << "Modified  " << modifiedFunctionNames.size() << "  function parameter lists " << endl;
    cout << "Found " << functionCallList.size() << " function call expressions " << endl;

    // Add parameter to function calls
    for (list<SgNode*>::iterator i = functionCallList.begin(); i != functionCallList.end(); i++) {
            SgFunctionCallExp* functionCallExp = isSgFunctionCallExp(*i);
            ROSE_ASSERT(functionCallExp != NULL);

            if(isSgFunctionRefExp(functionCallExp->get_function()) ){
                SgFunctionSymbol * symbol = isSgFunctionRefExp(functionCallExp->get_function())->
                                    get_symbol_i();
                assert(symbol!=NULL);

                char * name = symbol->get_name().str();

                // Only add the parameter if we have modified the corresponding function

                if(modifiedFunctionNames[name] == 1){
                        // Add the address of the struct to the function call arguments
                        functionCallExp->append_arg(lookupTLStruct(functionCallExp));
                }
            }
            else {
```

```
                cerr << "We don't yet handle function call by pointers" <<
                " SgPointerDerefExp is, I can probably append the function parameter(file=" <<
                 functionCallExp->get_file_info()->get_filenameString() << ")" <<  endl;

            }
    }

}



/** Build a list of TL Variable References */
list<SgVarRefExp*> buildListOfTLVariableReferences ( SgNode* node ) {

  // return variable
  list<SgVarRefExp*> TLVariableUseList;

  // list of all variables (then select out the global variables by testing the
  // scope, or the statics by the appropriate manner)
  list<SgNode*> nodeList = NodeQuery::querySubTree ( node, V_SgVarRefExp );

  list<SgNode*>::iterator i = nodeList.begin();
  while(i != nodeList.end())
        {
                        SgVarRefExp *variableReferenceExpression = isSgVarRefExp(*i);

                        assert(variableReferenceExpression != NULL);

                        assert(variableReferenceExpression->get_symbol() != NULL);
                        assert(variableReferenceExpression->get_symbol()->get_declaration() != NULL);
                        assert(variableReferenceExpression->get_symbol()->get_declaration()->get_scope()
                                                                != NULL);

            SgInitializedName* variableName = variableReferenceExpression->get_symbol()->
                                                get_declaration();
                        SgScopeStatement* variableScope = variableName->get_scope();

                                assert(variableScope);

                        // Check if this is a variable declared in global scope, if so, then save it
                        if (isSgGlobal(variableScope) != NULL)
                                TLVariableUseList.push_back(variableReferenceExpression);

            // Note that variableReferenceExpression->get_symbol()->get_declaration() returns the
            // SgInitializedName not the SgVariableDeclaration where it was declared!
            SgVariableDeclaration* variableDeclaration = isSgVariableDeclaration(variableName->
                                                get_parent());

            // Also save any static variables
                        if(variableDeclaration != NULL && isVarDeclStatic(variableDeclaration)) {
                                TLVariableUseList.push_back(variableReferenceExpression);
            }

          i++;
        }

  return TLVariableUseList;
}


SgClassDeclaration* createTLClass(SgGlobal* scope) {


  // We have two options for inserting the class declaration. The first is simple, but is buggy

#if USE_MIDDLELEVELREWRITE
  // This mechanism is not yet working. I have submitted a bug report for it
  char newCode[256];
  sprintf(newCode, "struct %s {void *%s;};\n", structName, placeHolderName);
  MiddleLevelRewrite::insert(scope, newCode,
                                                        MidLevelCollectionTypedefs::StatementScope,
                                                        MidLevelCollectionTypedefs::TopOfCurrentScope);

#else
   Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
   assert(fileinfo != NULL);
    SgClassDefinition *classDefinition = new SgClassDefinition(fileinfo);
    assert(classDefinition != NULL);

    SgClassDeclaration *classDeclaration =
                new SgClassDeclaration(fileinfo, structName, SgClassDeclaration::e_struct, NULL,
                                                                                        classDefinition);
    assert(classDeclaration != NULL);
    // Set the defining declaration in the defining declaration!
    classDeclaration->set_definingDeclaration(classDeclaration);
    classDeclaration->set_scope(scope);
    classDeclaration->set_parent(scope);
    // Set the end of construct explictly
    // (where not a transformation this is the location of the closing brace)
    classDefinition->set_endOfConstruct(fileinfo);
```

```
    SgClassDeclaration* nondefiningClassDeclaration =
                  new SgClassDeclaration(fileinfo,structName,SgClassDeclaration::e_struct,NULL,NULL);
    assert(classDeclaration != NULL);
    // Set the internal reference to the non-defining declaration
    classDeclaration->set_firstNondefiningDeclaration(nondefiningClassDeclaration);
    // Set the defining and no-defining declarations in the non-defining class declaration!
    nondefiningClassDeclaration->set_firstNondefiningDeclaration(nondefiningClassDeclaration);
    nondefiningClassDeclaration->set_definingDeclaration(classDeclaration);
    nondefiningClassDeclaration->setForward();
    nondefiningClassDeclaration->set_scope(classDeclaration->get_scope());
    classDefinition->set_declaration(classDeclaration);
    classDefinition->set_parent(classDeclaration);

    //  Insert a place_holder variable for convenience in later calls
    //  Do not delete this, as it is actually used to insert variables after itself.
    //  For some reason the rewrite mechanism will allow insertion of statements after other
    //  statements, but this requires an existing statement to use
    SgName *varname = new SgName(placeHolderName);
    SgVariableDeclaration *variable = new SgVariableDeclaration(fileinfo, *varname, new SgTypeInt);
    variable->set_parent(classDefinition);
    varname->set_parent(variable);
//     varname->set_scope(variable->get_scope());
    classDeclaration->get_definition()->append_member(variable);


    // Insert the class declaration before the first declaration in a non-header file

    list<SgDeclarationStatement*> decls = scope->get_declarations();
    list<SgDeclarationStatement*>::iterator d;

     for(d=decls.begin();d != decls.end();++d){
         // Insert after typedef statements in a non-header file
          if(! isStatementInHeader(*d) && !isSgTypedefDeclaration(*d)){
                insertStmt(*d, classDeclaration, true);
                break;
            }
      }

#endif


   // Find the SgClassDeclaration for this newly inserted class/struct
   // This is here if we choose some other higher level method for rewriting/inserting the struct
   // Additionally it is a nice sanity check
    SgClassDeclaration *newClassDeclaration=NULL;
    list<SgDeclarationStatement*>::iterator i = scope->get_declarations().begin();
    while(i != scope->get_declarations().end())
         {
           SgClassDeclaration *classDeclaration = isSgClassDeclaration(*i);
           if (classDeclaration != NULL && strcmp(structName, classDeclaration->get_name().str())==0)
                 newClassDeclaration = classDeclaration;
           i++;
         }
    assert(newClassDeclaration);

  return newClassDeclaration;
}


 void copyTLClassToFile(SgClassDeclaration *classDeclaration, SgFile* file){

    // Insert the class declaration before the first declaration in a non-header file
     assert(classDeclaration != NULL);
     SgNode* n = classDeclaration->copy(SgTreeCopy());
     SgClassDeclaration *classDeclarationDeepCopy = isSgClassDeclaration(n);

     list<SgDeclarationStatement*> decls = file->get_globalScope()->get_declarations();
     for(list<SgDeclarationStatement*>::iterator d=decls.begin();d != decls.end();++d){
             // Insert after typedef statements in a non-header file
             if(! isStatementInHeader(*d) && !isSgTypedefDeclaration(*d)){
                     insertStmt(*d, classDeclarationDeepCopy, true);
                     break;
             }
     }

}



SgFunctionDeclaration* createTLInitializer(SgGlobal* scope, SgDeclarationStatement* TLStruct) {
     assert(TLStruct != NULL);
     assert(scope != NULL);

     Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
     assert(fileinfo != NULL);

     // Create new function which will be inserted as a member function of class c
     SgBasicBlock *bb = new SgBasicBlock(fileinfo,NULL);
     SgFunctionDefinition *funcdef = new SgFunctionDefinition(fileinfo,bb);
     bb->set_parent(funcdef);
```

```
    // Setup new function
    SgReferenceType* intPointerType = new SgReferenceType(new SgTypeInt);
    SgName n("unused :("); // this gets ignored and the name comes from somewhere else
    SgInitializedName *name = new SgInitializedName(n,intPointerType,0,0,0);
    name->set_scope(funcdef);
    SgFunctionParameterList *fpl = new SgFunctionParameterList(fileinfo);
    fpl->append_arg(name);
    name->set_parent(fpl);

    SgFunctionType *func_type = new SgFunctionType(new SgTypeVoid(), FALSE);

    SgName func_name(initFuncName);
    SgFunctionDeclaration* fdecl= new SgFunctionDeclaration(fileinfo, func_name, func_type, funcdef);
    funcdef->set_declaration(fdecl);

    insertStmt(TLStruct, fdecl, false);

    return fdecl;
}


/** Put each TL variable into the TL struct and remove its original declaration */
void moveTLVariablesIntoClass (list<SgInitializedName*> & globalVariables,
                               list<SgInitializedName*> & staticVariables,
                               SgClassDeclaration* classDeclaration )
{

    // Remove all duplicates from the global variable list
    removeDuplicates(globalVariables);


    SgVariableSymbol* globalClassVariableSymbol = NULL;
    Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

    //==========================================================
    // Add in the globals
    for (list<SgInitializedName*>::iterator var = globalVariables.begin();
                var != globalVariables.end(); var++){



#ifdef DEBUG
            cout << "Adding variable " << (*var)->get_name().getString() << " to struct" << endl;;
#endif

            SgVariableDeclaration* globalVariableDeclaration = isSgVariableDeclaration((*var)->
                                                               get_parent());
            assert(globalVariableDeclaration != NULL);

            // Get the global scope from the global variable directly
            SgGlobal* globalScope = isSgGlobal(globalVariableDeclaration->get_scope());
            SgInitializedName *in =  *(globalVariableDeclaration->get_variables().begin());
            assert(globalScope != NULL);

            SgVariableDeclaration *firstVarInStruct = isSgVariableDeclaration(*(classDeclaration
                                                      ->get_definition()->getDeclarationList().begin()));

            // there should be at least some placeholder or real variables in this struct
            assert(firstVarInStruct!=NULL);



            /*

                2 options here:

                    1) Create a null expression and move comments onto it

                    2) move comments onto next declaration
                        This should work since there will probably never be a global
                        variable at the bottom of a file,
                        and even if there is, its attached comments won't likely be missed :)


            */

            // Reassociate any comments/preprocessor statements to the next declaration

            // find next declaration
            list<SgDeclarationStatement*> & decllist =  globalScope->get_declarations();
            SgDeclarationStatement *ds = globalVariableDeclaration;
            list<SgDeclarationStatement*>::iterator i = find(decllist.begin(), decllist.end(), ds);
            SgDeclarationStatement *followingDecl=NULL;
             if(i != decllist.end() && (i++)!=decllist.end()){ // If there is a following declaration
                    followingDecl = *i;
                    reassociatePreprocessorDeclarations(globalVariableDeclaration, followingDecl);
             }
            else { // If there is no following declaration. We should probably attach to previous one
                    cerr << "ERROR: not yet implemented: reassociating comments when no " <<
                    "declaration is after the global in a file" << endl;
            }
```

```
            // As usual, the higher level mechanisms(here LowLevelRewrite), don't work
            // Here the insertion just doesn't happen??? not sure why yet
            // Remove the variable
#if 1
            // This one doesn't detach comments and preprocessor stuff before moving
            // Thus an include statement could move into the class :(
            SgStatement *parent = isSgStatement(globalVariableDeclaration->get_parent());
            assert(parent);
            parent->remove_statement(globalVariableDeclaration);
#else
            LowLevelRewrite::remove(globalVariableDeclaration);
#endif


            // Fixup the variable to make sure it is output, and not clinging to its original file
            globalVariableDeclaration->set_file_info(fileinfo);



            // Add the variable to the class
#if 1
            classDeclaration->get_definition()->append_member(globalVariableDeclaration);
            globalVariableDeclaration->set_parent(classDeclaration->get_definition());
#else
        SgDeclarationStatementPtrList &decllist = classDeclaration->get_definition()->get_members();
        SgDeclarationStatement* firstDecl = *decllist.begin();
        insertStmt (firstDecl, globalVariableDeclaration, FALSE);
#endif

        }

    //=========================================================
    // Add in the static variables
    // Statics can be global or not global
    // If global, they have already been moved into the struct, and we should not attempt to move
    // them again. We should however rename them with some mangled scope containing a filename

    for (list<SgInitializedName*>::iterator var = staticVariables.begin();
                var != staticVariables.end(); var++){
            Sg_File_Info *fileinfo=(*var)->get_file_info();

            // If this is a global variable, then don't move it into the struct again
            if(find(globalVariables.begin(),globalVariables.end(),*var) == globalVariables.end()){
                SgVariableDeclaration* staticVariableDeclaration = isSgVariableDeclaration((*var)
                                                                    ->get_parent());
                assert(staticVariableDeclaration != NULL);

                SgVariableDeclaration *firstVarInStruct = isSgVariableDeclaration(*(classDeclaration->
                get_definition()->getDeclarationList().begin()));

                // there should be at least some placeholder or real variables in this struct
                assert(firstVarInStruct);
                // TODO: See note for globals above
                SgStatement *parent = isSgStatement(staticVariableDeclaration->get_parent());
                assert(parent);
                parent->remove_statement(staticVariableDeclaration);
                classDeclaration->get_definition()->append_member(staticVariableDeclaration);
                assert(staticVariableDeclaration->get_parent());
            }

            // Remove the static classifier
            SgDeclarationStatement * decl = (*var)->get_declaration();
            SgDeclarationModifier &declModifier = decl->get_declarationModifier();
            SgStorageModifier &storageModifier = declModifier.get_storageModifier();
            storageModifier.setDefault();
            assert(! storageModifier.isStatic());

            // Replace name with mangled name
            // TODO: simplify these if possible
            (*var)->set_name((*var)->get_mangled_name());
        }
    cout << endl;
    return;
}

void fixupReferencesToTLVariables ( list<SgVarRefExp*> & variableReferenceList)
{

// Now fixup the SgVarRefExp to reference the global variables through a struct
for (list<SgVarRefExp*>::iterator var = variableReferenceList.begin();
                var != variableReferenceList.end(); var++)  {
        assert(*var != NULL);

        SgNode* parent = (*var)->get_parent();
        assert(parent != NULL);

        // If this is not an expression then is likely a meaningless statement such as ("x;")
        SgExpression* parentExpression = isSgExpression(parent);
        assert(parentExpression != NULL);
```

```
        // Build the reference through the global class variable ("x" --> "AMPI_globals.x")

        // Build source position information (marked as transformation)
        Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
        assert(fileinfo != NULL);

        // Lookup the correct struct to use for the redirected reference
        SgExpression *lhs = lookupTLStruct(*var);
        assert(lhs);

        // Build "AMPI_globals.x" from "x"
        SgArrowExp* redirectedReference = new SgArrowExp(fileinfo,lhs,*var);

        assert(redirectedReference != NULL);

        if (parentExpression != NULL)  {
                // Introduce reference to *var through the data structure

                // case of binary operator
                SgUnaryOp* unaryOperator = isSgUnaryOp(parentExpression);
                if (unaryOperator != NULL) {
                        unaryOperator->set_operand(redirectedReference);
                }
                else  {
                        // case of binary operator
                        SgBinaryOp* binaryOperator;
                        SgExprListExp* exprListExp;
                        SgAssignInitializer* assignInitializer;
                        SgConditionalExp* condExp;
                        SgSizeOfOp* sizeofExp;
                        if (binaryOperator = isSgBinaryOp(parentExpression)){
                                // figure out if the *var is on the lhs or the rhs
                                if (binaryOperator->get_lhs_operand() == *var)  {
                                        binaryOperator->set_lhs_operand(redirectedReference);
                                }
                                else  {
                                        assert(binaryOperator->get_rhs_operand() == *var);
                                        binaryOperator->set_rhs_operand(redirectedReference);
                                }
                        }
                        else if(exprListExp = isSgExprListExp(parentExpression)){
                                // Where the variable appears in the function argument list the
                                // parent is a SgExprListExp
                                exprListExp->replace_expression (*var, redirectedReference);
                                // FIXME: This call is deprecated
                        }
                        else if(assignInitializer = isSgAssignInitializer(parentExpression)){
                                assignInitializer->set_operand(redirectedReference);
                        }
                        else if(condExp = isSgConditionalExp(parentExpression)){
                                // The trinary operator ?:
                                if (condExp->get_conditional_exp() == *var)  {
                                        condExp->set_conditional_exp(redirectedReference);
                                }
                                else if (condExp->get_true_exp() == *var)  {
                                        condExp->set_true_exp(redirectedReference);
                                }
                                else if (condExp->get_false_exp() == *var)  {
                                        condExp->set_false_exp(redirectedReference);
                                }
                                else {
                                        assert(0 && "some conditional operator broken");
                                }

                        } else if(sizeofExp = isSgSizeOfOp(parentExpression)){
                                if (sizeofExp->get_operand_expr() == *var)  {
                                        sizeofExp->set_operand_expr(redirectedReference);
                                }
                                else {
                                    assert(0 && " sizeof operator broken" );
                                }
                        }
                        else  {
                                // ignore these cases for now!
                                switch(parentExpression->variantT()) {
                                        case V_SgInitializer:
                                        case V_SgRefExp:
                                        case V_SgVarArgOp:
                                        default: {
                                                cerr << "BAD Error: default reached in "
                                                        "switch  parentExpression = " << endl;

                                                }
                                        }
                        }
                }
            }
        }
}
```

```
void transformTLVariablesToUseStruct ( SgProject *project )
{
    // Call the transformation of each file (there are multiple SgFile
    // objects when multiple files are specfied on the command line!).
    assert(project != NULL);

    // These are the global variables in the input program (provided as helpful information)
    list<SgInitializedName*> globalVariables = buildListOfGlobalVariables(project);
    list<SgInitializedName*> staticVariables = buildListOfStaticVariables(project);

#ifdef DEBUG
    cout << "Project Wide: global variables: " << endl;
    for (list<SgInitializedName*>::iterator var = globalVariables.begin(); var != globalVariables.end();
                                                                                var++)
        cout << "    " << (*var)->get_name().str() << endl;
    cout << endl;

    cout << "Project Wide: static variables: " << endl;
    for (list<SgInitializedName*>::iterator var = staticVariables.begin(); var != staticVariables.end();
                                                                                var++){
        cout << "    " << (*var)->get_name().str() << " ; mangled =" <<
                                                (*var)->get_mangled_name().str() << endl;
        Sg_File_Info *fileinfo=(*var)->get_file_info();
        cout << "          was in file " << fileinfo->get_filenameString() << endl;
    }
    cout << endl;
#endif

    SgFilePtrList* fileList = project->get_fileList();
    SgFilePtrList::iterator file;

    SgClassDeclaration* classDeclaration ;
    SgFunctionDeclaration* TLInit ;

    // First handle the main file
    bool foundFileMain = false;
    for(file=fileList->begin(); file != fileList->end(); ++file)  {

        if( overwrite_existing_files )
            (*file)->set_unparse_output_filename( (*file)->get_sourceFileNameWithPath() );


        if(isFileMain(*file)){
            cout << "Found main in file " << (*file)->get_file_info()->get_filenameString() << endl;

            assert(foundFileMain == false); // should only be one main file
            foundFileMain = true;

            // get the global scope within the file
            SgGlobal* globalScope = (*file)->get_globalScope();
            assert(globalScope != NULL);

            // Build the class declaration
            Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
            assert(fileinfo != NULL);
            classDeclaration = createTLClass(globalScope);

            // Create the initializer function
            TLInit = createTLInitializer(globalScope,classDeclaration);

            // Find references to TL variables
            list<SgVarRefExp*> variableReferenceList = buildListOfTLVariableReferences(project);

    #ifdef DEBUG
            cout << "TL variables referenced in this project(all files)" << endl;
            for (list<SgVarRefExp*>::iterator var = variableReferenceList.begin();
            var != variableReferenceList.end(); var++)
                cout << "    " << (*var)->get_symbol()->get_declaration()->get_name().str() ;
            cout << endl;
    #endif

            // Put the global variables into the class, removing the original declarations
            moveTLVariablesIntoClass(globalVariables,staticVariables,classDeclaration);

            // Create the struct on the stack and add it as a parameter to
            // all function calls and declarations
            declareClassAndInitInMain(globalScope, classDeclaration, TLInit);

            addTLClassAsParameter(project, classDeclaration);

            // Fixup all references to Thread Local variable to access the variable through the class
            // ("x" --> "AMPI_struct.x")
            fixupReferencesToTLVariables(variableReferenceList);

            // move all initializers into class from the variables that
            fixupInitializers(classDeclaration, TLInit);
        }
    }

    if(foundFileMain == false) {
```

```
            cerr << "ERROR: Didn't find a file containing main() or similar. "
            "We currently must have such a file" << endl;
            return;
        }


    // Cleanup struct by removing the temporaray place holder variable declaration
    fixupClassDeclarationPlaceHolder(classDeclaration);

    // Cleanup on each file
    for(file=fileList->begin(); file != fileList->end(); ++file) {
            // Put the struct definition in the top of all files
            if(!isFileMain(*file)){
                    cout << " filename= " << (*file)->getFileName() << endl;
                    copyTLClassToFile(classDeclaration, *file);
            }

            // Remove any "extern" definitions from the global scope of this file
            SgGlobal* globalScope = (*file)->get_globalScope();
            assert(globalScope != NULL);
            list<SgDeclarationStatement*> decls = globalScope->get_declarations();
            list<SgDeclarationStatement*>::iterator i;
            for (i=decls.begin(); i!=decls.end(); ++i) {
                    SgVariableDeclaration *varDecl = isSgVariableDeclaration(*i);
                    if (varDecl != NULL && isVarDeclExtern(varDecl)) {
                                    globalScope->remove_statement(varDecl);
                    }
            }
    }


}

// ****************************************
//              MAIN PROGRAM
// ****************************************
int main( int argc, char * argv[] )
{
    cout << "==========================================================" << endl;
    cout << "Thread Local Variable Encapsulation Translator. \nIf last command line argument is "
                    "\"-OVERWRITE\" then the source code will be overwritten" << endl;

    if(string(argv[argc-1]) == string("-OVERWRITE")){
            cout << " I am OVERWRITING all your source files :) " << endl << endl;
            overwrite_existing_files = true;
    } else {
            overwrite_existing_files = false;
    }

    cout << "==========================================================" << endl;

    // Build the AST used by ROSE
    // Strip off last parameter"-OVERWRITE" if we have it
    SgProject* project = frontend(overwrite_existing_files?argc-1:argc, argv);
    assert(project != NULL);

    // transform application as required
    transformTLVariablesToUseStruct(project);

#if 1
    generateDOT(*project);
    generatePDF(*project);
#endif

    // Code generation phase (write out new application "rose_<input file name>")
    return backend(project);
}
```

Appendix B

Automated Performance Tracing: Code Listing

The following is the source code for the Automated Performance Tracing-Call Insertion tool described in Chapter 4. The main source file for this translator is `globalVariableRewrite.C`. Additionally the shared code described in Appendix D is required.

```
/*

Insert performance profiling calls into AMPI codes.
We would like better profiling of MPI codes with Projections.

Author: Isaac Dooley

*/


#include <iostream>
#include <algorithm>
#include <string>
#include <rose.h>

#include <globalVariableCommon.h>


using namespace std;

/** Build a list of functions within a specified call depth from main
    depth=0 means just main will be included
    depth=1 means just main or functions called by main will be included
    depth=2 means those in depth=1 and any called by them is in the list

    Obviously it is hard to do much with function pointers, so we ignore them.
*/
list<SgFunctionDeclaration*> buildFunctionDefsToDepth(int depth, SgProject *project){
    list<SgFunctionDeclaration*> funcDeclList;

    list<SgNode*> funcDecls = NodeQuery::querySubTree (project,V_SgFunctionDeclaration);
    for (list<SgNode*>::iterator i = funcDecls.begin(); i != funcDecls.end(); i++) {
        if(isFunctionMain(isSgFunctionDeclaration(*i))){
            funcDeclList.push_back(isSgFunctionDeclaration(*i));
        }
    }


    for(int d=0;d<depth;d++){

        list<SgFunctionDeclaration*> newFuncs;
        // for each function in the list
        for (list<SgFunctionDeclaration*>::iterator i = funcDeclList.begin(); i != funcDeclList.end();
                    i++) {
            // find any SgFunctionRefExp inside the functions
            list<SgNode*> childRefs = NodeQuery::querySubTree (*i,V_SgFunctionRefExp);
            for (list<SgNode*>::iterator i = childRefs.begin(); i != childRefs.end(); i++) {
                SgFunctionDeclaration *fdecl = isSgFunctionRefExp(*i)->get_symbol()->get_declaration ();
                 newFuncs.push_back(fdecl);
            }


        }

        newFuncs.sort();
        funcDeclList.merge(newFuncs);
        removeDuplicates(funcDeclList);
```

```
        cout << " Found " << funcDeclList.size() << " functions to trace at depth " << d+1 << endl;

    }

    return funcDeclList;
}


void insertTimerCalls(SgBasicBlock *bb, SgStatement *begin, SgStatement *end){

    SgStatement * begincopy = isSgStatement(begin->copy(SgTreeCopy()));
    SgStatement * endcopy = isSgStatement(end->copy(SgTreeCopy()));
    assert(begincopy != NULL && endcopy != NULL);

    bb->prepend_statement(begincopy);

    // don't append to main since main always has a return statement(albeit implicitly sometimes)
    assert(bb != NULL);
    assert(isSgFunctionDefinition(bb->get_parent()) != NULL);
    SgFunctionDeclaration *fdecl = isSgFunctionDefinition(bb->get_parent())->get_declaration();
    assert(fdecl!=NULL);
    if(!isFunctionMain(fdecl))
        bb->append_statement(endcopy);


    // also insert it before any return statements

    list<SgNode*> returns = NodeQuery::querySubTree (bb,V_SgReturnStmt);
    for (list<SgNode*>::iterator i = returns.begin(); i != returns.end(); i++) {
        endcopy = isSgStatement(end->copy(SgTreeCopy())); // make a copy to insert
        insertStmt (isSgStatement(*i), endcopy, true);
    }

}


void insertTimerCalls(SgFunctionDefinition *func,
                      SgFunctionDeclaration *beginFuncDecl,
                      SgFunctionDeclaration *endFuncDecl,
                      SgFunctionDeclaration *regFuncDecl,
                      SgFunctionDeclaration *mainFunc){

    Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

    // First create the call to traceBeginFuncProj
    SgFunctionSymbol *startFuncSymbol = new SgFunctionSymbol(beginFuncDecl);
    // TODO: Probably not correct, but it works for now
    SgFunctionType *startFuncType = new SgFunctionType(new SgTypeVoid());
    SgFunctionRefExp *startFunctionRefExp =
        new SgFunctionRefExp(fileinfo, startFuncSymbol, startFuncType);
    SgExprListExp *startArgList = new SgExprListExp(fileinfo);
    SgStringVal *startparam1 = new SgStringVal(fileinfo, func->get_declaration()->get_name().str() );
    startArgList->append_expression(startparam1);
    SgStringVal *startparam2 = new SgStringVal(fileinfo, "UnknownFile" );
    startArgList->append_expression(startparam2);
    SgIntVal *startparam3 = new SgIntVal(fileinfo, 1 );
    startArgList->append_expression(startparam3);
    SgFunctionCallExp *startFuncCall =
        new SgFunctionCallExp(fileinfo, startFunctionRefExp, startArgList, startFuncType);
    SgExpressionRoot *startExprRoot =
        new SgExpressionRoot( fileinfo, startFuncCall, startFuncType);
    SgExprStatement *startStatement = new SgExprStatement(fileinfo, startExprRoot);

    // Second create the call to traceEndFuncProj
    SgFunctionSymbol *endFuncSymbol = new SgFunctionSymbol(endFuncDecl);
     // TODO: Probably not correct, but it works for now
    SgFunctionType *endFuncType = new SgFunctionType(new SgTypeVoid());
    SgFunctionRefExp *endFunctionRefExp = new SgFunctionRefExp(fileinfo, endFuncSymbol, endFuncType);
    SgExprListExp *endArgList = new SgExprListExp(fileinfo);
    SgStringVal *endparam1 = new SgStringVal(fileinfo, func->get_declaration()->get_name().str() );
    endArgList->append_expression(endparam1);
    SgFunctionCallExp *endFuncCall =
        new SgFunctionCallExp(fileinfo, endFunctionRefExp, endArgList, endFuncType);
    SgExpressionRoot *endExprRoot =
        new SgExpressionRoot( fileinfo, endFuncCall, endFuncType);
    SgExprStatement *endStatement = new SgExprStatement(fileinfo, endExprRoot);

    SgBasicBlock* body = func->get_body ();
    if (body && !body->get_statements().empty())
        {
            insertTimerCalls(body, startStatement, endStatement);
        }


    // Finally insert the register call to the top of main
    SgFunctionSymbol *regFuncSymbol = new SgFunctionSymbol(regFuncDecl);
    // TODO: Probably not correct, but it works for now
    SgFunctionType *regFuncType = new SgFunctionType(new SgTypeVoid());
    SgFunctionRefExp *regFunctionRefExp = new SgFunctionRefExp(fileinfo, regFuncSymbol, regFuncType);

    SgExprListExp *regArgList = new SgExprListExp(fileinfo);
```

```
    SgStringVal *param1 = new SgStringVal(fileinfo, func->get_declaration()->get_name().str() );
    regArgList->append_expression(param1);
    SgIntVal *param2 = new SgIntVal(fileinfo, -999 );
    regArgList->append_expression(param2);

    SgFunctionCallExp *regFuncCall = new SgFunctionCallExp(fileinfo, regFunctionRefExp, regArgList,
        regFuncType);
    SgExpressionRoot *regExprRoot = new SgExpressionRoot( fileinfo, regFuncCall, regFuncType);
    SgExprStatement *regStatement = new SgExprStatement(fileinfo, regExprRoot);

    SgBasicBlock* mainbody = mainFunc->get_definition()->get_body();
    if (mainbody && !mainbody->get_statements().empty())
        {
            mainbody->prepend_statement(regStatement);
        }

}



// Insert timer calls at a node if it is a function definition
void insertTimerCalls (SgProject* project, int tracedepth)
{
    SgFunctionDeclaration *beginFuncDecl=NULL, *endFuncDecl=NULL, *regFuncDecl=NULL, *mainFuncDecl=NULL;

    // Find function to insert a corresponding SgFunctionCallExp for
    list<SgNode*> functions = NodeQuery::querySubTree (project,V_SgFunctionDeclaration);
    for (list<SgNode*>::iterator i = functions.begin(); i != functions.end(); i++) {
        SgFunctionDeclaration *func_decl = isSgFunctionDeclaration(*i);
        char *fun_name = func_decl->get_name().str();
        if(strcmp(fun_name,"traceBeginFuncProj")==0){
            beginFuncDecl = func_decl;
        }
        if(strcmp(fun_name,"traceEndFuncProj")==0){
            endFuncDecl = func_decl;
        }
        if(strcmp(fun_name,"traceRegisterFunction")==0){
            regFuncDecl = func_decl;
        }
        if(isFunctionMain(func_decl) ){
            mainFuncDecl = func_decl;
        }
    }

    if(beginFuncDecl==NULL){
        cerr << "FATAL ERROR: couldn't find function called " << "traceBeginFuncProj" << endl;
        return;
        }
    if(endFuncDecl==NULL ){
        cerr << "FATAL ERROR: couldn't find function called " << "traceEndFuncProj" << endl;
        return;
        }
    if(regFuncDecl==NULL){
        cerr << "FATAL ERROR: couldn't find function called " << "traceRegisterFunction" << endl;
        return;
        }
    if(mainFuncDecl==NULL){
        cerr << "FATAL ERROR: couldn't find a main or AMPI_main function" << endl;
        return;
        }




    // Insert timer calls at top and bottom of all functions
    list<SgNode *> funcs = NodeQuery::querySubTree (project,
                               V_SgFunctionDefinition);


    list<SgFunctionDeclaration*> functionsToTrace = buildFunctionDefsToDepth(tracedepth, project);

    for(list<SgFunctionDeclaration*>::iterator f = functionsToTrace.begin(); f!=functionsToTrace.end();
        ++f) {
            cout << "examining a function declaration" << endl;

      insertTimerCalls((*f)->get_definition(),beginFuncDecl,endFuncDecl,regFuncDecl,mainFuncDecl);


      }

}

int main ( int argc, char * argv[] )
{
    int tracedepth;

    cout << "=================================================" << endl;
    cout << "Function Tracing insertion tool.\n Use -tracedepth n " << endl;
```

```
    if(string(argv[argc-2]) == string("-tracedepth")){
            cout << " Tracing to depth " << argv[argc-1] << endl << endl;
            tracedepth = atoi(argv[argc-1]);
    } else {
            tracedepth=4;
    }
    cout << "==========================================" << endl;


  SgProject* project = frontend (argc, argv);
  ROSE_ASSERT(project != NULL);

  insertTimerCalls(project, tracedepth);

#if 1
     generateDOT(*project);
//      generatePDF(*project);
#endif

  return backend (project);
}
```

Appendix C

Automatic PUP Function Creation: Code Listing

The following is the source code for the Automated PUP Creation tool described in Chapter 5. The main source file for this translator is `insertPUPs.C`.

```
/*
Created by Isaac Dooley

Purpose: create pup routines for classes

How:

    For each class defined in the file
        Create list of all class variables(private or public)
        Create a new function declaration, and insert it into the class
        add a pup statement for each class variable

Variations:
    Do we pup all class variables?
    Do we pup inherited variables?

Notes:
    We will use the MIDDLE LEVEL REWRITE mechanism . This mechanism still has a
    number of bugs, e.g. multifile support.

*/

#include <rose.h>
#include <list>
#include <vector>
#include <iostream>
#include <exception>
#include <string>

using namespace std;


void insert_PUP_In_Class(SgClassDefinition* c, SgType *pupperType, SgFunctionRefExp* pupfunction) {
/* MIDDLELEVEL doesn't work since:
            1) classes are unsupported as nodes into which you can insert codeA
            2) it creates a file, inserts a string and then parses it with EDG. This file
                doesn't include the headers which define the PUP namespace
*/

Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();


// Create new function which will be inserted as a member function of class c
SgBasicBlock *bb = new SgBasicBlock(fileinfo,NULL);
SgFunctionDefinition *funcdef = new SgFunctionDefinition(fileinfo,bb);
bb->set_parent(funcdef);

// Setup parameters for new function
SgReferenceType* pupperPointerType = new SgReferenceType(pupperType);
SgName n("p");
SgInitializedName *puppername = new SgInitializedName(n,pupperPointerType,0,0,0);
puppername->set_scope(funcdef);
SgFunctionParameterList *fpl = new SgFunctionParameterList(fileinfo);
fpl->append_arg(puppername);
puppername->set_parent(fpl);

// Setup the SgMemberFunctionDeclaration which pulls the function into the class
SgCtorInitializerList *ctor = new SgCtorInitializerList(fileinfo);
SgType * returntype = new SgTypeVoid();
SgFunctionType * functype = new SgFunctionType(returntype, false);
SgName mfdname("PUP");
```

```
SgMemberFunctionDeclaration *mfd = new SgMemberFunctionDeclaration(fileinfo,mfdname,NULL,funcdef);
mfd->set_parent(c);
mfd->set_CtorInitializerList (ctor);
mfd->set_parameterList(fpl);
mfd->set_type(functype);
mfd->set_scope(c);
fpl->set_parent(mfd);
ctor->set_parent(mfd);
funcdef->set_parent(mfd);
funcdef->set_declaration(mfd);

// Finally add the member function to the class
c->append_member(mfd);


// Insert a pup call for each member variable of the class
list<SgDeclarationStatement*> &members = c->get_members();
cout << "There are " << members.size() << " declarations in the class " << endl;
for(list<SgDeclarationStatement*>::iterator i=members.begin(); i!=members.end();++i){
    SgVariableDeclaration *vardecl;
    if(vardecl=isSgVariableDeclaration(*i)){

        SgInitializedNamePtrList &args = vardecl->get_variables();
        assert(args.size() == 1);
        // assume that the front=only variable being declared is the one we want
        SgInitializedName *finalArg = args.front();
        assert(finalArg);
        SgVariableSymbol *variableSymbol = new SgVariableSymbol(finalArg);
        assert(variableSymbol);

        cout << "found a variable member "  << variableSymbol->get_name().getString() <<  endl;

        //The pupper itself
        SgVariableSymbol* pupvarsymbol = new SgVariableSymbol(puppername);
        SgVarRefExp* pupvar = new SgVarRefExp(fileinfo, pupvarsymbol);

        // The variable to be pupped
        SgVarRefExp* varToPup = new SgVarRefExp(fileinfo, variableSymbol);
        SgClassSymbol *cs = new SgClassSymbol(c->get_declaration());
        SgThisExp *t = new SgThisExp(fileinfo, cs, 0);
        SgArrowExp *a = new SgArrowExp(fileinfo,t,varToPup,variableSymbol->get_type());

        SgExprListExp* exprlist = new SgExprListExp(fileinfo);
        exprlist->prepend_expression(a);
        exprlist->prepend_expression(pupvar);

        SgFunctionCallExp* pfc = new SgFunctionCallExp(fileinfo, pupfunction, exprlist, pupperType);


        // Put a in a null expression statement
        SgExpressionRoot *exprRoot = new SgExpressionRoot( fileinfo, pfc, pupperType );
        SgExprStatement * exprStatement = new SgExprStatement(fileinfo, exprRoot);

        bb->append_statement(exprStatement);



    }

}

}


void transformClassesAddPUPs(SgProject* project){
        bool done = false;

        SgNamedType *pupperType;
        list<SgNode*> types = NodeQuery::querySubTree (project,V_SgType);
        for(list<SgNode*>::iterator i=types.begin();i!=types.end() && !done;++i){

            if(isSgNamedType(*i)){
                const string qname = isSgNamedType(*i)->get_qualified_name().getString();
                const string dname = "PUP::er";

                if(qname == dname){
                    cout << "Found some type called PUP::er, hopefully I got the right one." << endl;
                    pupperType = isSgNamedType(*i);
                    done=true;
                }
            }
        }


        // We first find a function declaration for our pup routine
        // then we derive an SgFunctionRefExp for it
        Sg_File_Info* fileinfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

        cout << "Looking for a pup function to reference" << endl;
        SgFunctionDeclaration *pupperfdecl=NULL;
```

```
        list<SgNode*> funcs = NodeQuery::querySubTree (project,V_SgFunctionDeclaration);
        cout << "created a list of " << funcs.size() << " function declarations" << endl;
        done=false;
        for(list<SgNode*>::iterator i=funcs.begin();i!=funcs.end() && !done;++i){
            SgFunctionDeclaration *fdecl;
            if(fdecl = isSgFunctionDeclaration(*i)){
                    const string qname = fdecl->get_name().getString();
                    const string dname = "operator|";

                    // Some will be called "operator|"
                    if(qname == dname){
                            cout << "Found some function declaration called " << qname << endl;
                            cout << "qualified name = " << fdecl->get_qualified_name().getString() << endl;

                            pupperfdecl = fdecl;

                    }


            }
        }
        cout << "done iterating through list " << endl;

        assert(pupperfdecl);

        SgFunctionSymbol *pupsymbol = new SgFunctionSymbol(pupperfdecl);
        SgFunctionType *pupfunctype = new SgFunctionType(pupperType);
        SgFunctionRefExp* pupfunction = new SgFunctionRefExp(fileinfo, pupsymbol, pupfunctype);


            list<SgNode*> classes = NodeQuery::querySubTree (project,V_SgClassDefinition);
            cout << "Number of classes appearing in this project: " << classes.size() << endl;
            for (list<SgNode*>::iterator i = classes.begin(); i != classes.end(); ++i ) {
                //Add PUP to this class
                SgClassDefinition* c = isSgClassDefinition(*i);
                if(c!=NULL)
                    insert_PUP_In_Class(c,pupperType, pupfunction);
                else
                    cout << "WARNING: NodeQuery::querySubTree (globalScope,V_SgClassDefinition) returned "
                            "something that is not a SgClassDefinition" << endl;
            }
}


// ****************************************
//              MAIN PROGRAM
// ****************************************
int main( int argc, char * argv[] )
{
    // Build the AST used by ROSE
    SgProject* project = frontend(argc,argv);
    assert(project != NULL);

    cout << "Automatic PUP creation translator" << endl;

    // transform application as required
    transformClassesAddPUPs(project);

    generateDOT(*project);
    generatePDF(*project);

    // Code generation phase (write out new application "rose_<input file name>")
    return backend(project);
}
```

Appendix D

Common Shared Routines: Code Listing

The following is the source code for functions used in multiple translators. This code is in a source file called `globalVariableCommon.C`.

```
/**
*   A set of useful routines for identifying global and static variables
*    Used by both globalVariableRewrite and globalVariableFind
*/


#include "globalVariableCommon.h"

using namespace std;


/** Determine if a statement is in a header file */
bool isStatementInHeader(SgStatement* s){
  char * filename = s->get_file_info()->get_filename();
  assert(filename);
  int len = strlen(filename);
  if(filename[len-1]=='h' && filename[len-2]=='.')
        return true;
  else
        return false;
}



/** Determine if a variableDeclaration is static or not */
bool isVarDeclStatic(SgVariableDeclaration *variableDeclaration){
        // TODO: a declaration has two flags which can specify statics. Add the other one as well
        assert(variableDeclaration != NULL);
        SgDeclarationModifier &declModifier = variableDeclaration->get_declarationModifier();
        SgStorageModifier &storageModifier = declModifier.get_storageModifier();
        return storageModifier.isStatic();
}

/** Determine if a variableDeclaration is extern or not */
bool isVarDeclExtern(SgVariableDeclaration *variableDeclaration){
        assert(variableDeclaration != NULL);
        SgDeclarationModifier &declModifier = variableDeclaration->get_declarationModifier();
        SgStorageModifier &storageModifier = declModifier.get_storageModifier();
        return storageModifier.isExtern();
}



list<SgInitializedName*> buildListOfStaticVariables (SgFile* file ) {
  // This function builds a list of static variables (from a SgFile).
  assert(file != NULL);

  // return variable
  list<SgInitializedName*> staticVariableList;

  SgNode* node = file;

  list<SgNode*> nodeList = NodeQuery::querySubTree ( node, V_SgVariableDeclaration );

  list<SgNode*>::iterator i = nodeList.begin();
  while(i != nodeList.end())
        {
          SgVariableDeclaration *variableDeclaration = isSgVariableDeclaration(*i);
          assert(variableDeclaration != NULL);

            if(isVarDeclStatic(variableDeclaration)){
```

```
                list<SgInitializedName*> & variableList = variableDeclaration->get_variables();
                list<SgInitializedName*>::iterator var = variableList.begin();
                while(var != variableList.end())
                  {
                        assert((*var)->get_scope());
                        staticVariableList.push_back(*var);
                        var++;
                  }

          } else {
            //                cout << "didn't find a static " << endl;
          }


          i++;

        }

  return staticVariableList;
}


list<SgInitializedName*> buildListOfGlobalVariables (SgFile* file ) {
  // This function builds a list of global variables (from a SgFile).
  assert(file != NULL);

  list<SgInitializedName*> globalVariableList;

  SgGlobal* globalScope = file->get_globalScope();
  assert(globalScope != NULL);
  list<SgDeclarationStatement*>::iterator i = globalScope->get_declarations().begin();
  while(i != globalScope->get_declarations().end())
        {
          SgVariableDeclaration *variableDeclaration = isSgVariableDeclaration(*i);
          if (variableDeclaration != NULL)
            if(! isVarDeclExtern(variableDeclaration))
                {

                  list<SgInitializedName*> & variableList = variableDeclaration->get_variables();
                  list<SgInitializedName*>::iterator var;
                  for(var=variableList.begin(); var != variableList.end(); ++var)
                        {
                            // Don't include various globals which come from rose somehow
                            // At one point one showed up called _April_12_2005
                            if(! (*var)->get_file_info()->isCompilerGenerated() ){
                                assert((*var)->get_scope());
                                globalVariableList.push_back(*var);
                            } else {
                                cerr << "WARNING: Ignoring compilerGenerated variable  " <<
                                                    (*var)->get_name().getString() << endl;
                            }

                        }
                }
          i++;
        }

  return globalVariableList;
}

list<SgInitializedName*> buildListOfGlobalVariables ( SgProject* project ) {

  // This function builds a list of global variables (from a SgProject).
  list<SgInitializedName*> globalVariableList;

  SgFilePtrList* fileList = project->get_fileList();
  SgFilePtrList::iterator file = fileList->begin();

  // Loop over the files in the project (multiple files exist
  // when multiple source files are placed on the command line).
  while(file != fileList->end())
        {
          list<SgInitializedName*> fileGlobalVariableList = buildListOfGlobalVariables(*file);
          fileGlobalVariableList.sort();
          globalVariableList.merge(fileGlobalVariableList);

          file++;
        }

  return globalVariableList;
}


list<SgInitializedName*> buildListOfStaticVariables ( SgProject* project ) {

  // This function builds a list of static variables (from a SgProject).
  list<SgInitializedName*> staticVariableList;

  SgFilePtrList* fileList = project->get_fileList();
  SgFilePtrList::iterator file = fileList->begin();
```

```
  // Loop over the files in the project (multiple files exist
  // when multiple source files are placed on the command line).
  while(file != fileList->end())
        {
          list<SgInitializedName*> fileStaticVariableList = buildListOfStaticVariables(*file);
          staticVariableList.merge(fileStaticVariableList);
          file++;
        }

  return staticVariableList;
}




void printGlobalsAndStatics(SgProject* project){

 // These are the global variables in the input program (provided as helpful information)
    list<SgInitializedName*> globalVariables = buildListOfGlobalVariables(project);
    list<SgInitializedName*> staticVariables = buildListOfStaticVariables(project);

    cout << "Project Wide: global variables: " << endl;
    for (list<SgInitializedName*>::iterator var = globalVariables.begin(); var != globalVariables.end();
    var++)
          cout << "    " << (*var)->get_name().str() << endl;
    cout << endl;

    cout << "Project Wide: static variables: " << endl;
    for (list<SgInitializedName*>::iterator var = staticVariables.begin(); var != staticVariables.end();
        var++){
          cout << "    " << (*var)->get_name().str() << " ; mangled =" << (*var)->get_mangled_name().str()
          << endl;
          Sg_File_Info *fileinfo=(*var)->get_file_info();
          cout << "          was in file " << fileinfo->get_filenameString() << endl;
    }
    cout << endl;

}



/** main-like functions are specified here */
bool isFunctionMain(SgFunctionDeclaration *func_decl){
    char *func_name = func_decl->get_name().str();
    if(strcmp(func_name,"main")==0){
        cout << "Main function called \"main\"" << endl;
        return true;
    }
  if(strcmp(func_name,"AMPI_Main")==0){         // This one will occur for AMPI C programs. See mpi.h
        cout << "Main function called \"AMPI_Main\"" << endl;
        return true;
    }
  if(strcmp(func_name,"AMPI_Main_cpp")==0){     // This one will occur for AMPI C++ programs. See mpi.h
        cout << "Main function called \"AMPI_Main_cpp\"" << endl;
        return true;
    }
  return false;
}


/** Insert a single statement at the given target node using the
 low-level rewrite interface, either before or after the target. */
void insertStmt (SgStatement* target, SgStatement* stmt, bool insertBefore)
{
  ROSE_ASSERT (target && stmt);
  SgStatementPtrList temp_stmt_list;
  temp_stmt_list.push_front (stmt);
  LowLevelRewrite::insert(target, temp_stmt_list, insertBefore);
}



/** Remove duplicate global variables from the list */
void removeDuplicates(list<SgInitializedName*> & globalVariables){
    list<SgInitializedName*> temp;
    list<SgInitializedName*>::iterator i,j;

    for(i=globalVariables.begin();i!=globalVariables.end();++i){
        bool found = false;
        // check if this is in the temp array yet. If not, insert it
        for(j=temp.begin();j!=temp.end();++j){
            if( (*i)->get_name().getString() == (*j)->get_name().getString() ){
                found=true;
            }
        }
        if(!found)
            temp.push_back(*i);
```

```
    }

globalVariables = temp;
}


void removeDuplicates(list<SgFunctionDeclaration*> & funcs){
    list<SgFunctionDeclaration*> temp;
    list<SgFunctionDeclaration*>::iterator i,j;

    for(i=funcs.begin();i!=funcs.end();++i){
        bool found = false;
        // check if this is in the temp array yet. If not, insert it
        for(j=temp.begin();j!=temp.end();++j){
            if( (*i)->get_name().getString() == (*j)->get_name().getString() ){
                found=true;
            }
        }
        if(!found)
            temp.push_back(*i);
    }

funcs = temp;
}
```

References

[1] Rose website. http://www.llnl.gov/CASC/rose/.

[2] The weather research&forecasting model website. http://wrf-model.org/.

[3] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[4] MILC Collaboration. Mimd lattice computation (milc) collaboration home page. http://www.physics.indiana.edu/∼sg/milc.html.

[5] Chao Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[6] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[7] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[8] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[9] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[10] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[11] Orion Lawlor, Milind Bhandarkar, and Laxmikant V. Kalé. Adaptive mpi. Technical Report 02-05, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[12] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation, 2003.

[13] Yunheung Paek and David A. Padua. Compiling for scalable multiprocessors with polaris. *Parallel Processing Letters*, 7(4):425–436, 1997.

[14] D Quinlan, S. Ur, and R. Vuduc. An extensible open-source compiler infrastructure for testing.

[15] Daniel Quinlan, Qing Yi, Gary Kumfert, Thomas Epperly, and Tamara Dahlgren. Toward the automated generation of components from existing source code.

[16] Markus Schordan, , and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *Lecture Notes in Computer Science: Proc. of Joint Modular Languages Conference (JMLC03)*, volume 2789, pages 214–223. Springer-Verlag, June 2003.

[17] W. C Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers. A description of the advanced research wrf version 2. Technical Report Technical Note NCAR/TN-468+STR, June 2005.