Instrumenting Memory Accesses in the Open Research Compiler
A Case Study of our CMPUT 680 Course Project

Kevin Andrusky and Stephen Curial
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{andrusky, curial}@cs.ualberta.ca

September 20, 2006

Abstract

This document is designed to help fill the void that exists because of the lack of publicly available documentation for the Open Research Compiler (ORC) and to help make modifying the ORC seem a less daunting process. The information contained in this document is based upon the authors' experiences while implementing their CMPUT 680 course project in the ORC. This document is available on Stephen Curial's web site[1], which also links to or contains all other ORC documentation that the authors have discovered.

[1] http://www.cs.ualberta.ca/~curial/ORC/index.html

Contents

1 Introduction

This document is not intended to be an authoritative source of documentation for the Open Research Compiler (ORC) but, rather, is intended to supplement the documentation that does exist. In particular, only those issues which arose during the implementation of our CMPUT 680 project are documented. Many parts of the ORC were not touched during our project work, and, as a result, are not documented herein.

This document is not a project report, nor is it a technical report. As a result, we have attempted to take a more personal approach in writing it (the pronoun 'you' appears frequently, for instance). Our goal is to make life as easy as possible for all those who rise to the challenge and attempt to wrangle the ORC into submission. In our opinion, losing the personal aspect of the writing detracts from that goal.

1.1 A Brief History of the ORC

One important source of information about the history of the ORC is the documentation in the global optimizer `osprey1.0/be/opt/opt_main.cxx`. Further, information compiled by Sebastian Pop [6], as well as Colin Cherry and Dan Lizotte [3] provides some insights into the ORC's past.

The ORC is a mature, highly-developed, industry-strong compiler. It has been developed for well over 10 years by many talented compiler writers at Silicon Graphics[2].

In 1994, the compiler group at SGI was developing the MIPSPro compiler (code name Ragnarok) for the MIPS R8000 processor. Most of the optimizations in the Ragnarok compiler were designed for scientific applications. In the second half of 1994, the compiler group decided to overhaul the compiler infrastructure to create a stable and efficient compiler that could be used in production environments for both scientific and non-scientific applications. In August 1994, the group started designing a new global optimizer (code named Mongoose). Implementation began in October of 1994.

In 1999 the compiler group at SGI believed that the MIPS series of processors were soon to be discontinued. As a result, they began to re-target their compiler for the IA-32 (Itanium) and later IA-64 (Itanium 2) architecture. This focus on the IA-64 architecture lead the compiler group at SGI to develop the SGIPro64 1.0 compiler (code named Osprey).

In late 2000, SGI released the Pro64 compiler to the public as the Open64 compiler, an open source project hosted on SourceForge[3]. In the fourth quarter of 2000, Intel, along with the Chinese Academy of Sciences, branched off from the Pro64 source, and started development of the Open Research Compiler[4] (the ORC). Version 1.0 of the ORC was released in January 2002 with a minor revision (v1.1) being released in July of 2002. The main differences between the ORC and the Pro64 stem from two years of development work by a core group of 15 - 20 engineers. The most

[2]SGI - http://www.sgi.com
[3]http://open64.sourceforge.net/
[4]http://ipf-orc.sourceforge.net/

significant change was a re-designed code generation phase. Version 2.0 of the ORC was released in January 2003, and, once again, a minor revision followed in July of that year.

1.2 Documentation Available for the ORC

There is not a lot of detailed documentation available for the ORC, but the symbol table and the intermediate-level code representation (WHIRL), two very important parts of the compiler, are well documented. It is strongly recommended that anyone wishing to alter or understand the ORC print out and read the WHIRL [7] and Symbol Table [8] documentation that SGI released with the Pro64 Compiler. Berube [2] also provides some documentation that may be useful if you are using feedback directed optimization in the ORC. Cherry and Lizotte [3] report on several optimizations in the ORC and how they can be controlled to improve the performance of the application. Specifically, Cherry and Lizotte study four benchmarks and detail how several commonly used optimizations affect the performance of those benchmarks. As well, several helpful tutorials are linked from the ORC web site.

1.3 What We Were Trying to Accomplish

While our work (on tree traversal classification) will likely be different from the work of anyone reading this document, we hope to transfer some of the knowledge we gained from designing and implementing our project in the ORC. Thus we will describe our project to allow the reader to compare our project with other possible projects or investigations.

Our project centered around trying to solve the following problem:

> Given the information that a pointer p is part of a tree data structure and that an access of p in code begins a pattern of memory accesses m representing a tree search or a tree traversal, determine what type of search or traversal is being used by the trace m started with the access to p. [1]

The information which follows is based upon our attempts to solve this problem. We broke our problem into two parts, in order to do as little work as possible inside the ORC. However, we were still left with the task of finding and instrumenting references to tree pointers inside the ORC, and the examples below are based on that work. We hope that, despite being specific to our problem, these examples will give you insight into how to solve your problems in the ORC.

2 Where to Start

Determining where to implement an optimization is a very important decision that depends on the task at hand. You will need to know what kind of information and data structures (eg. dominance

tree, CFG, etc.) your optimization requires and what information is available to you at the various phases within the compiler. Before choosing where to implement the optimization, you must become familiar with what happens when the ORC is run.

2.1 What Happens When The ORC is Run

Basically, when you execute the ORC you are actually running a driver which executes a number of other programs. You can see which programs the ORC calls if you use the `-v` flag when running the ORC. Figure 1 (originally from [4]) shows the applications that are called by the driver, along with the data flow between those applications.

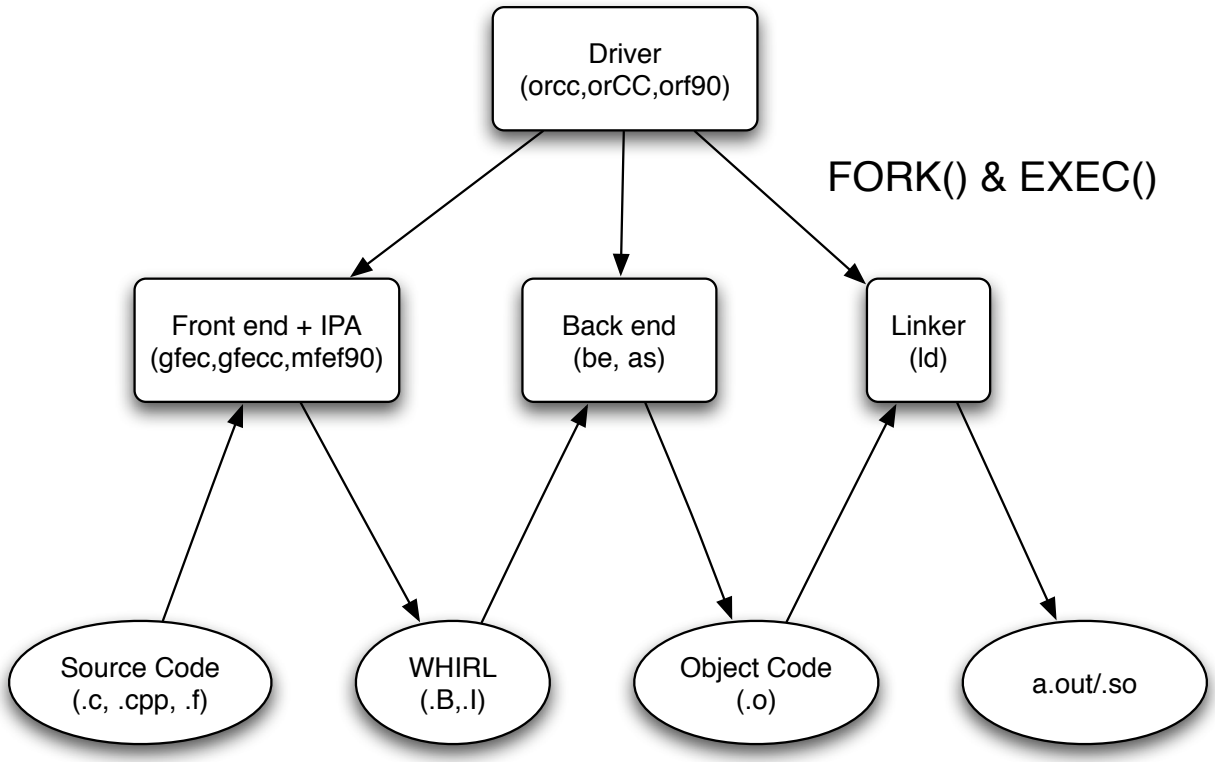

Figure 1: Compilation Model in the ORC [4].

2.2 Where We Implemented Our Framework

We decided that our framework should be implemented in High WHIRL[5] within the global optimizer. (ie. `osprey1.0/be/opt/`) While our project wasn't related to global optimizations, we decided to implement it there because of the data structures that are readily available in the global optimizer. Some of the data structures that we had available for use in the global optimizer were

[5]See page 5 of the SGI WHIRL Specification [7].

the WHIRL tree, the symbol table, the dominator and post-dominator trees, and the SSA, among many others.

It is definitely worth noting that there is a significant drawback to gaining the use of these data structures. If you wait until later in the global optimizer to perform your optimization or alterations to WHIRL, then you might have to manually alter the many data structures that the ORC uses. Failure to do so correctly may result in the ORC crashing during later optimizations or code generation, or, worse, may result in the ORC generating incorrect code.

3 Important Source Files in the ORC

The ORC is composed of approximately 6500 source files (based upon a count of .h and .cxx files in the osprey1.0 directory tree inside the ORC distribution) comprising around 1.2 million lines of source code. Needless to say, it is often a daunting task to find the relevant function calls and data structures for your project. You may also wish to find possible sources of code which could be used as a template for your own work. While the files that are of importance to anyone working with the ORC are dependent on the project, there are some files which are likely to be important to just about anyone interested in working with or modifying the ORC data structures. These files are detailed below, and many will be referenced again in the sections which follow.

All paths are relative, given that the current working directory is `orc/src/`.

- `osprey1.0/common/com/symtab.h`: Contains the functions needed to create a new symbol table entry and a new type table entry. Also includes the necessary functions to initialize the values in the symbol and type table.
- `osprey1.0/common/com/symtab_defs.h`: Contains the definitions of the symbol table and all associated types.
- `osprey1.0/common/com/symtab_utils.h`, and `osprey1.0/common/com/symtab_access.h`: Contains the symbol table accessor functions, type table accessor functions, and field table accessor functions. Also includes functions to print various structures to the terminal.
- `osprey1.0/common/com/wn.h`: Contains a large number of functions which allow for the creation of specific types of WHIRL nodes. This is very helpful if you are attempting to add code to a program at the intermediate-language level. Contrary to what you might be inclined to believe, given its name, it does not contain the definition of a WN (WHIRL node) structure.
- `osprey1.0/common/com/wn_util.h`: Contains a wide variety of functions for getting values out of a WHIRL node, as well as for modifying the information currently stored in a WHIRL node. The file structure is quite confusing, with macro definitions, and methods which are overloaded (the only difference in the method signature being a `const` on one of the parameters). Fortunately, it is generally easy to see, from the names of the functions and macros, what each does. `wn_util.h` is an excellent file (as it contains source, as well as function

prototypes) in which to see how a WHIRL node can be manipulated, especially if there is no function for the manipulation you wish to perform.

- `osprey1.0/common/com/wn_core.h`: Contains the actual structure definition of a WHIRL node. If you need to modify a part of a WHIRL node that doesn't seem to have a modification function in `wn_util.h` or if you are just interested in the structure of a WHIRL node, this is the place to look.
- `osprey1.0/be/opt/opt_main.cxx`: Contains the primary functions in the global optimizer that call each specific optimization. `opt_main.cxx` is the file in which we inserted the function call to our tree-instrumentation function.
- `osprey1.0/be/opt/opt_tree.cxx`: The file we created in which to implement our instrumentation code.

4 Creating a Flag to Control the Optimizer

4.1 Files Modified

- `osprey1.0/common/com/config_opt.cxx`
- `osprey1.0/common/com/config_opt.h`
- `osprey1.0/driver/opt_actions.c`
- `osprey1.0/driver/OPTIONS`
- `osprey1.0/be/opt/opt_main.cxx`

4.2 General Procedure

For our project, we decided the first thing we needed was a flag that could be used to control whether or not the tree traversal profiling would be performed. Since we decided that our framework should be called in High WHIRL, and that a reasonable place to call our framework-creation method is `osprey1.0/be/opt/opt_main.cxx`, we needed some method of determining if our optimization was specified on the command line. Unfortunately, you can not simply create a global variable in the main function of the driver (`osprey1.0/driver/main.c`) because the driver forks and executes several programs, such as the back end.

We felt that the flag for our instrumentation code on should be part of the *opt* group and that *gen_tree_prof* seemed like a reasonable name. Thus our project code could be enabled by invoking the ORC with the `-OPT:gen_tree_prof` flag.

To create a flag of your own, you will need to modify the `void Process_Opt_Group ( string opt_args )` function that can be found in `osprey1.0/driver/opt_actions.c`. If you determine that the flag for your optimization has been specified you'll need to call `add_option_seen(`

O_gen_tree_prof). O_gen_tree_prof is defined in the header file osprey1.0/driver/option_names.h that is automatically generated from osprey1.0/driver/OPTIONS. Thus you will need to add a -gen_tree_prof flag to the file osprey1.0/driver/OPTIONS.

Next, you'll need to add extern BOOL Tree_Prof to osprey1.0/common/com/config_opt.h and BOOL Tree_Prof = FALSE to osprey1.0/common/com/config_opt.cxx. Finally, you'll need to modify static OPTION_DESC Options_OPT[] in osprey1.0/common/com/config_opt.cxx. The OPTION_DESC structure is defined in osprey1.0/common/util/flags.h.

Now you can use the variable Tree_Prof in osprey1.0/be/opt/opt_main.cxx to determine whether -OPT:gen_tree_prof was specified at the command line.

5 Playing with the WHIRL Tree (In the Global Optimizer)

5.1 Traversing the WHIRL Tree

You might think that traversing the WHIRL tree *should* be a relatively simple exercise. In osprey1.0/common/com/wn_util.h there exist several iterators that *should* allow you to traverse the WHIRL tree. We called our framework from a function in opt_main.cxx that was passed a pointer to the WHIRL tree (WN *wn_tree). We assumed that we could simply use an iterator to traverse this tree. As you have probably already determined, that was not the case.

That particular WHIRL tree (wn_tree) is broken during some of the optimizations and is not subsequently repaired. Fortunately, the ORC designers hid a non-broken version of the WHIRL tree inside the compiler. This tree, the one which should be used for traversing, can be obtained by calling comp_unit->Input_tree(). The WHIRL node iterators can be used with this tree to allow you, the programmer, to easily create a traversal. The WHIRL tree obtained from comp_unit->Input_tree() is passed to later stages of the compiler.

An example of how to traverse the WHIRL tree is given in Figure 2. It should be noted that the traversal is pre-order. Nodes are obtained before any of their children, and children are visited from the first child to the last. In the case of blocks (which do not have children, but rather a linked list of WHIRL nodes) nodes are visited from the first element of the list, to the last, and the subtree of each node in the list is processed fully before moving on to the next node.

5.2 Identifying Pointers to Tree Data Structures

To profile memory we first needed to be able to identify which pointers access tree data structures. Ghiya and Hendren [5] present a static analysis that can be used to identify pointer-based tree data structures in memory. Due to the limited time frame for our project we did not implement Ghiya and Hendren's framework, but instead recognize tree based data structures through the use of programmer hints. We assumed that, at a later stage, we could substitute our method for that of Ghiya and Hendren.

```
void traverse_whirl_tree(WN *wn_tree)
{
  WN_ITER *wn_itr;  /* WHIRL Tree Iterator*/
  OPCODE opcode;
  OPERATOR opr;
  WN *wn_tmp;

    for (wn_itr=WN_WALK_TreeIter(wn_tree);  wn_itr;
                                    wn_itr=WN_WALK_TreeNext(wn_itr)){
       wn_tmp = WN_ITER_wn(wn_itr);  //get this WHIRL node
       opcode = WN_opcode (wn_tmp);  //get the WHIRL node opcode
       opr = WN_operator(wn_tmp);    //get the WHIRL node operator
       /* Do whatever you wish for this node in the WHIRL tree */
    }
 }
```

Figure 2: Example of how to traverse the WHIRL tree.

5.2.1 Accessing the Symbol Table

Accessing the symbol table to find the name of (or any other relevant information for) the identifier in a given WHIRL node *should* be straight forward. Using the symbol table access methods should allow you to extract the identifier's name simply by using a access method: `str = ST_name(WN_st(wn))` where `wn` is a WHIRL node (`WN *`).

Unfortunately, the global optimizer creates an auxiliary symbol table and destroys part of the primary symbol table. This is confirmed by the comments on line 1292 of `opt_main.cxx` that hint that the string table is freed, and that using the regular `dump_tree()` function will cause a segmentation fault.

```
// create aux symbol table
// cannot print WHIRL tree after this point, use dump_tree_no_st
```

All of the information in the original symbol table (and much more) can be found in the auxiliary (aux) symbol table. To access this table, you need to access members of the COMP_UNIT class. Using the accessor functions you can simply get the string representing the identifier's name as follows: `str = comp_unit->Opt_stab()->St_name(WN_aux(wn));` . If you want to print the WHIRL tree at this point you can call `fdump_tree_no_st(stdout, wn)`.

When we discovered that the symbol table and WHIRL tree are broken in the global optimizer we decided to move where our function was called. We moved our function to immediately before the WHIRL tree and symbol table are broken and the auxiliary data structures are created. This was a perfectly acceptable solution for us since we did not need to use any of the data structures that are computed in the global optimizer. If you do not need to use all of the data structures created by

the optimizer, we strongly recommend moving your optimization as far up in the ORC compilation process as you can.

5.3 Modifying the WHIRL Tree

We found that if we changed the tree obtained from `comp_unit->Input_tree()`, that we could change the code that was generated. Thus we knew that our changes could take place on the tree pointed to by `comp_unit->Input_tree()`. This was a very satisfying result, because, otherwise, we would have been forced to move our own alterations to the WHIRL tree to some point before the original WHIRL tree is broken. Although we discovered we could move our code above the point where the comp_unit method was needed, you might not be so fortunate.

Once we had confirmed that the changes that we made to the WHIRL tree were propagated through to code generation it was time to actually modify the WHIRL tree. The first thing we needed to do for our project was to identify what memory accesses we wanted to dereference. In our course report we describe how we identify tree pointer dereferences [1]. The code to identify these dereferences was an extension of the code fragment provided as Figure 2.

5.3.1 Inserting Code

To insert code you must find the block that immediately dominates the pointer dereference that is being instrumented. This is the block where you want to insert code into. As well you must also identify the child before which you want to insert code. We accomplished both of these tasks by creating a function that mapped a WHIRL node to its parent. Once we had this mapping we could easily find the block to insert code into and the child of that block that we want to insert code before (in our case, this was the first child). The `osprey1.0/be/be/wn_outlining.cxx` file had a function that created a map between WHIRL nodes and we modified that function to create our parent map.

Once we had identified where we wanted to insert our code, the next step was to create the WHIRL nodes to insert. Figure 3 shows an example of how to insert a node into the WHIRL tree. Note that `get_parent_block()` and `get_first_kid_of_block()` are functions that we created. The WHIRL node to instrument is `wn_to_instrument`, and is the only input necessary for this code fragment.

We wanted to call a library function, so we inserted a VCALL node rather than the comment node that was inserted in Figure 3. Inserting a call to a dynamic library is very similar to inserting a call to any other function. We created a VCALL as the functions we were were inserting didn't return anything (ie. they had a void return type). To create a VCALL you must create a symbol table entry for the function with the function name. Then you have to create PARAM nodes that, in our case, represented the the children of a tree node. Our instrumentation function took as a parameter the addresses of the children of the structure that was being accessed. We discuss how we identified child pointers in our course project report [1].

```
WN *parent_block = get_parent_block(wn_to_instrument);
WN *first_kid = get_first_kid_of_block(wn_to_instrument,parent_block);

char *comment = "Inserting a comment into  WHIRL!";
WN *wn_comment = WN_CreateComment (comment);
WN_INSERT_BlockAfter( parent_block, first_kid, wn_comment);
```

Figure 3: Sample code inserting a comment into WHIRL

6 Pitfalls

Many things tripped us up as we were working on modifying the ORC. Some have been mentioned above, others have not been mentioned before in this document. In order to help others who modify the ORC, we fell that it is important to mention all of these pitfalls in one common section, in the hopes that, even if the rest of the document is of little use to you, at least you won't stumble over the same things in the ORC that we did.

- During the course of optimization, the ORC destroys the initial symbol table, and constructs and auxiliary symbol table for use instead. This can cause no end of frustration for someone who is attempting to make use of the symbol table while performing an optimization. If you believe that the symbol table you are using is not valid, you may wish to use the symbol table from the COMP_UNIT in the global optimizer.
- As with the Symbol Table, the WHIRL tree inside the global optimizer is destroyed during optimization. An auxiliary WHIRL tree can be found inside the COMP_UNIT from the global optimizer and we have confirmed that changes made to this WHIRL tree will be propagated through the optimizer and be included in the final compiled code.
- For the purposes of WHIRL output (and, possibly for including as debug information inside the final object code) a line number can be attached to any of the nodes in the WHIRL tree. However, great care must be taken when assigning line numbers to WHIRL nodes being inserted into the WHIRL tree. Incorrect line numbers can cause the optimizer to crash, or can break the WN_INSERT functions. We did not actually figure out exactly how line numbers break WHIRL, but we did determine that it is acceptable in WHIRL to not include line numbers at all for a node.
- As noted above, if you perform your optimization late in the optimization process, you might need to add additional information to the various data structures in the ORC. For example, the original placement of our optimization inside the ORC meant that, when a function was created, we had to create a symbol table entry for that function, and had to alter a structure of mu and chi lists. While altering the symbol table was not complicated, we eventually gave up on figure out how to properly create the lists, and moved our optimization to a point before those lists are created. Now we could count on the ORC to build those lists properly for us.

References

[1] K. Andrusky and S. Curial. Profiling tree traversals in the ORC. http://www.cs.ualberta.ca/∼curial/ORC/.

[2] Paul Berube. *Open Research Compiler: Feedback Directed Optimization*. Department of Computing Science, University of Alberta.

[3] Colin Cherry and Dan Lizotte. *Matching Optimizations to Code Characteristics in the Open Research Compiler*. Department of Computing Science, University of Alberta, December 2003.

[4] Guang Gao, J. Nelson Amaral, James C. Dehnert, and Ross A. Towle. Tutorial on the SGI Pro64 compiler infrastructure. In *PACT 2000: Int'l Conf. on Parallel Architectures and Compilation Techniques*, Philadelphia, Pennsylvania, October 2000.

[5] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–15, St. Petersburg, Florida, January 1996.

[6] Sebastian Pop. Sebastian Pop's web site at INRIA. http://www-rocq.inria.fr/∼pop/.

[7] Silicon Graphics, Inc. *CHAPTER 1 - WHIRL Intermediate Language Specification*, May 2000.

[8] Silicon Graphics, Inc. *CHAPTER 2 - WHIRL Symbol Table Specification*, May 2000.

Index