DRAGON2011: OPENUH-BASED SCALABLE VISUALIZATION TOOL FOR PROGRAM ANALYSIS

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Amrita Banerjee
December 2011
DRAGON2011: OPENUH-BASED SCALABLE
VISUALIZATION TOOL FOR PROGRAM ANALYSIS

Amrita Banerjee

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Zhigang Deng
Dept. of Computer Science

David Baldwin
Petroleum Geo-Services USA, Inc

Dean, College of Natural Sciences and Mathematics
Acknowledgements

I would like to dedicate this work to my advisor Dr Barbara Chapman. I am deeply indebted to her for accepting me as a part of HPCTools group and giving me the opportunity to explore on-going research in this field. I am thankful to her for her guidance and belief in me to develop Dragon2011.

Next major thanks goes to Dr Zhigang Deng and David Baldwin, who very generously agreed to serve on my thesis committee. Many thanks to them for taking their time out to read my thesis and for suggesting new directions to work on.

I would like to personally thank all members of the HPCTools group. Thank you for sharing your knowledge and fostering wonderful ideas for Dragon2011. I extend my profound thanks to Wei for his patience and support that helped me overcome many crisis situations and made this thesis possible. Warm regards to Sunita, Deepak, Dunni, Sayan, Priyanka and Tony for their encouragement and support.

I would like to take this opportunity to thank all my teacher’s and professors who play a vital role in my life. I would like to thank my family, friends, and dear roommates - Manisha, Anushaya and Pallavi for always boosting my confidence.

Finally, I would like to express my deepest gratitude to Ma-Baba(parents), Dadiya-Didan-Thamma-Dadu(grandparents), Chotu, Bholda and Amit. They had envisioned this success two and a half years back when even I did not believe in myself. No words can express the contribution they have made towards my career and my life. The work in this thesis is as much theirs as is mine and my advisor’s.
DRAGON2011: OPENUH-BASED SCALABLE
VISUALIZATION TOOL FOR PROGRAM ANALYSIS

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Amrita Banerjee
December 2011
Abstract

The size and complexity of real-world application programs can cause challenges when maintaining, parallelizing, upgrading or porting them. In such scenarios, application developers benefit from a structural overview of the code and program details which will help them analyze data usage, control flow and detect code regions profitable for parallelization. Few tools are able to provide such information especially in case of the FORTRAN programs.

Dragon2011 is an interactive system with a powerful Graphical User Interface (GUI) providing a range of information about the structure of source program in a graphical browser-based form, at the level of detail desired. It currently handles programs written in Fortran, C/C++ and OpenMP. The features supported include call graph, control flow graph and array region analysis. The necessary advanced program analysis capabilities are provided by the OpenUH research compiler, maintained by researchers of the HPCTools group at the University of Houston. Various phases of the OpenUH compiler extracts static as well as dynamic source code information. The latter is enabled by invoking the compiler’s feedback components.

Dragon2011’s predecessor tool, Dragon, had severe drawbacks such as poor layout algorithm, a read-only code browser and cluttering of graphs on the user’s screen. It was developed on MOTIF/LESSTIF with X11 which restricted the tool’s usage to Unix platforms only. In contrast, Dragon2011 is supported on Windows and Linux platforms. It inherits a scalable graphical layout algorithm from Graphviz. Additional features include a code editor with syntax highlighting, real-time search functionality and a detachable graph dock. Dragon2011 has been tested on large codes including GenIDLEST CFD application.
# Contents

1 Introduction .............................................. 1
   1.1 Contributions ......................................... 5
   1.2 Organization of Thesis ................................. 6

2 Role of the OpenUH compiler ............................... 7
   2.1 Overview ................................................. 7
   2.2 OpenUH Intermediate Representation .................. 10
       2.2.1 Overview ............................................. 10
       2.2.2 Components of WHIRL ............................... 11
       2.2.3 WHIRL Symbol Table ............................... 14
   2.3 Inter Procedural Analysis .............................. 17
   2.4 The Loop Nest Optimizer .............................. 20
       2.4.1 Automatic Parallelization ....................... 23
   2.5 The Global Optimizer .................................. 25
   2.6 Code Generation ....................................... 28
   2.7 Summary ................................................. 30

3 Dragon2011 - Architecture and Compiler Knowledge Extraction 31
   3.1 Motivation .............................................. 32
   3.2 Architectural Modifications ........................... 36
3.3 Compiler Extraction Modules ........................................... 40
  3.3.1 Call Graph ...................................................... 40
  3.3.2 Control Flow Graph ............................................. 42
3.4 Compilation Guidelines for using Dragon2011 ....................... 44
3.5 Summary .............................................................. 47

4 Dragon2011 - Design and Implementation ............................. 48
  4.1 Understanding Dragon2011 GUI-related classes .................. 48
  4.2 Using Dragon2011 .................................................. 52
    4.2.1 Basic Operations .............................................. 52
    4.2.2 Viewing Call Graph .......................................... 62
    4.2.3 Viewing Control Flow Graph ................................ 64
    4.2.4 Viewing Array Region Analysis data ........................ 65
  4.3 Evaluation ......................................................... 69
  4.4 Summary ............................................................. 71

5 Related Work ............................................................ 75
  5.1 Compiler-based Tools .............................................. 75
  5.2 Automatic Parallelizer ............................................. 79
  5.3 Performance Analysis Tools ....................................... 79
  5.4 Code Transformation Tools ....................................... 81

6 Conclusion and Future Work ........................................... 83
  6.1 Conclusion ......................................................... 83
  6.2 Future Work ....................................................... 84

Bibliography ............................................................... 86
# List of Figures

2.1 The OpenUH Compiler/Runtime Infrastructure .................................. 8  
2.2 WHIRL Symol Table produced by the front-ends ............................... 15  

3.1 Dragon version1.1 Architecture .................................................. 36  
3.2 Dragon2011 Architecture ......................................................... 37  
3.3 OpenUH Inter Procedural Analysis modules .................................... 40  

4.1 Dragon2011 - Main Classes ....................................................... 49  
4.2 Dragon2011 - Welcome Screen ................................................. 53  
4.3 Example demonstrating loading Dragon2011 project file .................... 54  
4.4 Dragon2011 - Main Window Elements ......................................... 55  
4.5 Call graph for NAS BT benchmark ............................................... 56  
4.6 Example demonstrating array region widget invocation ....................... 56  
4.7 Example demonstrating Code Browser syntax highlighting feature .......... 58  
4.8 Example demonstrating find functionality ..................................... 59  
4.9 Example demonstrating Grep Browser displaying results ..................... 60  
4.10 Detachable Graph Dock ............................................................ 61  
4.11 Understanding call graph widget components .................................. 63  
4.12 Control flow graph for NAS BT benchmark ................................... 64  
4.13 Example displaying control flow graph saving format ......................... 66  
4.14 Array region analysis for NAS BT benchmark .................................. 68
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Functionality of different OpenUH modules</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Components of WHIRL</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Layout of the WHIRL node</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Classification of Symbol Tables</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Purpose of Symbol Tables</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>WOPT Phases</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Qt Image Format Support</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Dragon2011 Classes Functionality</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>Control Flow Graph - Node Representation</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

High Performance Computing (HPC) applications are designed to optimize the parallel nature of the high-performance computational computing systems. They are naturally complex and usually involve usage of Application Programming Interfaces (APIs) such as OpenMP and Message Passing Interface (MPI) which provide flexibility in parallelizing code. The OpenMP API [46] provides portable, scalable model for developing parallel applications and supports shared-memory programming in C/C++ and Fortran. The API is easy to understand however it could become difficult for users to analyze their programs and insert OpenMP directives appropriately in them. In case of high-performance OpenMP programs, the user has to carefully examine the programs to specify complex attributes for their variables and insert barriers. The user has to apply program transformations to their programs to enhance parallelization.

An important aspect of the evolution of these applications is the challenge given
to the state of technology for empirical performance evaluation. The increasing complexity of parallel and distributed systems demands a performance tool or compiler to provide robust performance observation capabilities. The compiler plays a very important role in recognizing optimization and parallelization techniques which can lead to substantial performance gain in the applications.

Large scientific applications like NCAR’s High-Order Method Modeling Environment (HOMME) [53], that is a petascale-capable high-order element-based conservative dynamical core developed on the cubed-sphere grid, comprises of thousands of functions with several callee and caller function relations. The compiler automatic optimization and parallelization techniques in such an instance might not be very effective and may demand manual intervention.

The size and complexity of large real-world application are a serious impediment in understanding and changing them. Software visualization is an appropriate solution this problem. The idea is to model some aspect of software as a graph, and present the graph as a drawing to make it easier to understand the model. Graphs are convenient for describing the data types, functions, variables, control structures, files and even bugs in source code programs, or the structure of finite state machines and grammars. They can be created from static analysis, dynamic traces, or other sources. The goal of this thesis work, is to design a graphical browsable compiler-based analysis tool which will enhance the speed and preciseness of large-scale source code navigation. We named our tool as Dragon2011. The OpenUH compiler was adapted to perform and export the required program analysis.

The OpenUH [13] compiler is a branch of the open source Open64 compiler
suite for C, C++, Fortran 95/2003, supporting the IA-64, IA-32, Opteron Linux ABI, and PTX generation for NVIDIA GPUs. It consists of the front-ends with support for OpenMP 3.0 and Coarray Fortran (CAF), optimization modules, back-end lowering phases for OpenMP and coarrays, portable OpenMP and CAF runtimes, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 modules. OpenUH may be used as a source-to-source compiler for other machines using the IR-to-source tools. [28]

The OpenUH infrastructure is modularized, with different components that interact via a common intermediate representation called the WHIRL (Winning Hierarchical Intermediate Representation Language). The five levels of WHIRL are known as; very high (VH), high (H), mid (M), low (L) and very low (VL). Our tool mainly focuses on the VH and H levels of WHIRL which preserve the high level control flow constructs such as loops. VHL and HL WHIRL can be translated back to the source code however this can incur minor loss in semantics. The information extraction modules mainly operate on the inter-procedural analysis (IPA) and global optimizer phases of OpenUH.

The Graphical User Interface (GUI) of Dragon2011 has been developed in Qt (version 4.7) [21]. Qt is a cross-platform application and UI development framework. Qt is open source and is actively maintained and supported by Qt Project group. Qt has been used in Autodesk Maya, Adobe Photoshop Elements, Skype, VLC media player, VirtualBox, and Mathematica, DreamWorks, Google, KDE, Walt Disney Animation Studios and several more notable projects. Writing in C++ gives the user great control and offers working with libraries like STL etc. Since majority
of the OpenUH compiler has been written in C and C++, we wanted to maintain
the C++ framework. Qt offers several features, documented very well here [18],
and the community maintains extensive documentation support [17]. It provides
a rich set of classes [19] which offer several predefined properties including signals
and slots [20], the central feature of Qt which gives it an edge over other frame-
works. Qt provides optimization flags which enable improving graphical rendering
e.g. QGraphicsView::CacheModeFlag allows users to specify whether all the paint-
ing should be directly done to the viewport or not and also allows to specify if the
background should be cached or not. Several features achieved in Dragon2011 are
results of complimentary features from Qt.

With the rich graphic objects generated by Qt, we wanted to use a visualization
software or library which would offer representing the structural information in the
form of scalable abstract graphs and networks. We chose the Graphviz library [33] to
compute the positions of our graphical elements. Graphviz is a collection of software
for viewing and manipulating abstract graphs. The core of Graphviz consists several
layouts that can be used via a C library interface, streambased command line tools,
graphical user interfaces and web browsers. It offers a wide assortment of graphi-
cal features and output formats. The former makes it possible to write high-level
programs for querying, modifying and displaying graphs. Some practical systems
that rely on Graphviz for software visualization are the Acacia, Doxygen, and Mono
static analysis systems, the Syntacs toolkit for Java compiler generation, the Spin
concurrent protocol analyzer and the Bugzilla bug tracking system originally cre-
ated for the Mozilla (Netscape) open source project. We believe that the underlying
algorithms in the library will support scalability of large scientific application code.

1.1 Contributions

The motivation behind this work came from the original design of existing Dragon tool [38, 11, 39]. This tool was developed in X11/Lesstif and provided call graph, flow graph, array region analysis and data dependency information. The tool however had several drawbacks in terms of its usability and visualization strategies. A compiler generates several useful information for the related source code but the challenge lies in the design of the user interface, in terms of, reading the information and presenting in a user-friendly way.

The first step in the development of this tool was to understand the information the compiler generated. All the files dumped were in binary format and hence was not human readable. We introduced new functions to read the data in the text format. Once we understood our database structure the next step was to come up with strategies to display them. We wanted to support the tool on several platforms unlike its predecessor, so we chose the Qt cross-platform application development framework. As the first step from GUI point of view, we developed the module for generating graphs using Graphviz library. We then designed the main frame of the Dragon2011 with the code editor and graph dock and menu items. Next we developed the program to read the compiler files (in binary format) and generate the corresponding graph. The graph zooming and panning, syntax highlighting and print features were added. The array region analysis display tables were added and
developing this module was faster because the corresponding source file was in text format.

Dragon2011 software has been tested for several languages including C/C++, FORTRAN and OpenMP. It has also been tested on various scales of application including stand-alone to large applications like GenIDLEST.

The High Performance Computing Tools (HPCTools) research group, part of the Department of Computer Science at the University of Houston, recently released a new version of the OpenUH compiler during the Supercomputing Conference 2011 in Seattle, Washington. The Dragon2011 source code has been merged with the new version of the compiler and was demonstrated during this event.

1.2 Organization of Thesis

The organization of this thesis is as follows: Chapter 2 introduces the specific phases of the OpenUH compiler which play an important role in extracting the information Dragon2011 displays. Chapter 3 provides a brief introduction to the existing work and describes the motivational factors for a new version of the tool. It also describes the architectural modifications made to the design of the tool. Chapter 4 gives a detailed description of the compiler knowledge extraction techniques and reports GUI implementation details followed by results achieved on evaluating Dragon2011 on two major benchmarks namely NAS BT and GenIDLEST. Chapter 5 describes related tools and Chapter 6 presents the conclusion and some ideas for new functionalities that could be implemented in the future.
Chapter 2

Role of the OpenUH compiler

The Dragon2011 tool is built on top of OpenUH compiler. In this chapter we briefly summarize the role of the main modules of the compiler which help in extracting the information displayed by Dragon2011.

2.1 Overview

The OpenUH compiler is a research compiler maintained by the HPCTools Group [9] at University of Houston. It is based on the Open64 [15] compiler suite. As mentioned here [28], Open64 originated from the SGI MIPSPro compiler for the MIPSRI0000 processor, and was open-sourced as Pro64 in 2000 under the GNU public license. The University of Delaware became the official host for the compiler, now called Open64, in 2001 and continue to host the project today. Over the past 10 years, there have been wide contributions from industry and research institutions. Intel and
the Chinese Academy of Sciences partnered early on to develop the Open Research Compiler (ORC) which implemented a number of code generator optimizations and improved support for the Itanium target. A number of enhancements and features from the QLogic PathScale compiler was also merged in, including support for an x86 back-end. OpenUH supports C, C++, Fortran 95/2003, with support for a variety of targets including x86 64, IA-64, and IA-32. It is a well-written, modularized, robust, state-of-the-art compiler with support for C/C++ and Fortran 77/90.

Figure 2.1: The OpenUH Compiler/Runtime Infrastructure
Figure 2.1 depicts an overview of the design of OpenUH compiler and runtime infrastructure. It consists of the front-ends with support for OpenMP 3.0 and Coarray Fortran (CAF), optimization modules, back-end lowering phases for OpenMP and coarrays, portable OpenMP and CAF runtimes, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 modules. OpenUH may be used as a source-to-source compiler for other machines using the IR-to-source tools.

Table 2.1 lists the main functionalities performed by main OpenUH modules. Each module performs different tasks that can be enabled/disabled based on the corresponding flags selected while compilation. The information is consistent with functionalities of various Open64 modules.

<table>
<thead>
<tr>
<th>Module</th>
<th>Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front-end</td>
<td>Parsing source code, translation to Very High Level WHIRL</td>
</tr>
<tr>
<td>Inliner</td>
<td>Inlines procedures</td>
</tr>
<tr>
<td>Inter Procedural Analysis</td>
<td>Inlines procedures, alias analysis, array section, code layout, procedure cloning, dead function/variable elimination, constant propagation, translation to High Level WHIRL</td>
</tr>
<tr>
<td>Loop Nest Optimizer</td>
<td>Loop fission, fusion, unrolling, jam, interchange, peeling, tiling, vector data prefetching, loop dependence analysis</td>
</tr>
<tr>
<td>Global Optimizer</td>
<td>Architecture neutral optimizations, based on SSA form. General SSA form transforma-</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Software pipelining, instruction scheduling, use of hyperblocks, register allocation, generating object code.</td>
</tr>
</tbody>
</table>
2.2 OpenUH Intermediate Representation

2.2.1 Overview

A compiler can translate a source program to machine level instructions or intermediate code (e.g. java bytecodes) or another source. The compilation process does not involve a one-step procedure but a gradual transition from the high level language constructs to the low level machine instructions. In between, there are different levels of Intermediate Representations (IR) generated. The closer an IR is to the source language, the higher is its level. As a corollary, the more an IR resembles the machine instructions, the lower is its level.

WHIRL (Winning Hierarchical Intermediate Representation Language) [51] is the IR for OpenUH compiler. Using a common IR enables OpenUH to support multiple languages and multiple processor targets. The different front-ends of OpenUH translate the different languages to WHIRL. OpenUH has a sophisticated back-end composed of multiple components: the inter-procedural analyzer (IPA) and optimizer (IPO), loop-nest optimizer (LNO), global scalar optimizer and code generator (CG). WHIRL serves as the common interface among all these components.

Adapting a common intermediate representation for as many phases of the compilation process has several advantages. In the compilation process, some optimization passes like constant propagation, dead code elimination, and various liveness problems, have to be re-applied at different times and in different components of the
compiler. With a common IR, a single implementation of an optimization pass is sufficient. Communication between the compilation phases is also easier, since they work under the same medium.

The five levels of WHIRL are very high (VH), high (H), mid (M), low (L) and very low (VL). VH and H comprise of high level control flow constructs like DO_LOOP, DO_WHILE and IF. VH and H can be translated back to C and Fortran source code via modules whirl2c, whirl2f and whirl2f90. This could however incur some loss of semantics.

2.2.2 Components of WHIRL

The front-end generates a WHIRL file which comprises of WHIRL instructions and WHIRL symbol tables [52]. The instruction part of the WHIRL file represents the program code, organized in program units (PUs). These PUs are the nodes of the flow graph that Dragon2011 generates. WHIRL instructions are linked up in strictly tree form and hence can also be referred as WHIRL tree. The WHIRL tree thus represents the control flow and expressions. Each PU is a single tree. The table 2.2 below summarizes the components of the WHIRL node.

In addition to the above list, there are several data types supported by WHIRL. A WHIRL node is represented by the struct WN which has minimum allocated size of 24 bytes. The table 2.3 gives the layout of struct WN.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>This field specifies the operation performed by the instruction. Operators in WHIRL can be divided into three categories: structured control flow, statements, and expression.</td>
</tr>
<tr>
<td>Result and Descriptor Types</td>
<td>Result type (res) gives the data type of the result of the operation. Descriptor type (desc) gives the data type of the operands.</td>
</tr>
<tr>
<td>Kid Pointers</td>
<td>These are non-BLOCK nodes or non-leaves nodes which contain pointers to their children in the kids array.</td>
</tr>
<tr>
<td>Next and Previous Pointers</td>
<td>These pointers link the children of a BLOCK node to the statement nodes. The first statement of a BLOCK has null previous field, and the last statement has null next field.</td>
</tr>
<tr>
<td>Offset</td>
<td>This refers to the offset fields associated with load, store and load-address opcodes. Note that in VH and H WHIRL, it is not legal to fold the offset fields which can result in incorrect address calculation and also affect loop nest optimizer to do data dependence analysis</td>
</tr>
<tr>
<td>Mapping Mechanism</td>
<td>Each WHIRL node contains a word-sized map_id that effectively maps to a row in the mapping table. The map_ids are unique only within each PU, and the map tables are organized on a per-PU basis in the WHIRL file.</td>
</tr>
<tr>
<td>Source Code Position</td>
<td>The 64-bit field linenum specifies source position information. The line number is stored in a 32-bit field while remaining 32 bits contain the file and column number.</td>
</tr>
<tr>
<td>Additional Fields</td>
<td>These are operator-specific fields like symbol table indices and type table indices.</td>
</tr>
</tbody>
</table>
## Table 2.3: Layout of the WHIRL node

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte - 16</td>
<td>prev</td>
<td>previous pointer</td>
</tr>
<tr>
<td>byte - 12</td>
<td>next</td>
<td>next pointer</td>
</tr>
<tr>
<td>byte - 8</td>
<td>linenum</td>
<td>source position information</td>
</tr>
<tr>
<td>byte 0</td>
<td>offset</td>
<td>offset for loads, stores, etc</td>
</tr>
<tr>
<td>byte 0</td>
<td>trip_est</td>
<td>estimated trip count for LOOP_INFO</td>
</tr>
<tr>
<td>byte 2</td>
<td>depth</td>
<td>loop nesting depth for LOOP_INFO</td>
</tr>
<tr>
<td>byte 4</td>
<td>st_idx</td>
<td>symbol table index</td>
</tr>
<tr>
<td>byte 0</td>
<td>elem_size</td>
<td>element size of ARRAY</td>
</tr>
<tr>
<td>byte 8</td>
<td>operator</td>
<td>WHIRL operator</td>
</tr>
<tr>
<td>byte 9 bit 0</td>
<td>res</td>
<td>result type</td>
</tr>
<tr>
<td>byte 9 bit 5</td>
<td>operator</td>
<td>number of kids for n-ary operators</td>
</tr>
<tr>
<td>byte 11 bit 3</td>
<td>desc</td>
<td>descriptor type</td>
</tr>
<tr>
<td>byte 12</td>
<td>map_id</td>
<td>index into map table</td>
</tr>
<tr>
<td>byte 16</td>
<td>kids[0]</td>
<td>kid 0 - first pointer for BLOCK</td>
</tr>
<tr>
<td>byte 20</td>
<td>kids[1]</td>
<td>kid 1 - last pointer for BLOCK</td>
</tr>
<tr>
<td>byte 16</td>
<td>const_val</td>
<td>64-bit integer constant</td>
</tr>
<tr>
<td>byte 24+n</td>
<td>kids[2+n]</td>
<td>the (2+n)th kid for n &gt;= 0</td>
</tr>
</tbody>
</table>
2.2.3 WHIRL Symbol Table

The above section described the various data structures associated with WHIRL nodes. WHIRL requires symbol table for compilation, optimization and storage efficiency. Figure 2.2 provides a pictorial overview of the WHIRL symbol table produced by the front-ends. WHIRL symbol table comprise of a series of tables which can be categorized as global part and local part. Another classification is by place i.e. front-end and back-end tables. Table 2.4 classifies the different symbol tables and the purpose of each symbol table has been summarized in table 2.5. The description pertains to what each entry in the corresponding table represents.

<table>
<thead>
<tr>
<th>Module</th>
<th>Functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common tables for global and local parts</td>
<td>ST_TAB, INITO_TAB, ST_ATTR_TAB</td>
</tr>
<tr>
<td>exclusive global symbol tables</td>
<td>PU_TAB, TY_TAB, FLD_TAB, TYLIST_TAB, ARB_TAB, TCON_TAB, BLK_TAB, INITV_TAB,</td>
</tr>
<tr>
<td></td>
<td>STR_TAB, TCON_STR_TAB</td>
</tr>
<tr>
<td>exclusive local symbol tables, organized by</td>
<td>LABEL_TAB, PREG_TAB</td>
</tr>
<tr>
<td>program units (PUs)</td>
<td></td>
</tr>
<tr>
<td>free internal tables for individual compiler</td>
<td>IPA_STR_TAB, WOPT_ST_TAB, BE_ST_TAB</td>
</tr>
<tr>
<td>components</td>
<td></td>
</tr>
</tbody>
</table>

The symbol tables serve the purpose of communicating information obtained by the front-ends with the back-end components during compilation. As mentioned in the tables above, there are additional internal tables which can be created and used by the back-end components to store additional new information.
Figure 2.2: WHIRL Symol Table produced by the front-ends
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST_TABLE</td>
<td>a symbol</td>
</tr>
<tr>
<td>ST_ATTR_TAB</td>
<td>additional attribute with an ST entry, mostly empty</td>
</tr>
<tr>
<td>INITO_TAB</td>
<td>an initialized global or static data object with an INITV entry</td>
</tr>
<tr>
<td>INITV_TAB</td>
<td>initial value of a scalar component of a data object</td>
</tr>
<tr>
<td>PU_TAB</td>
<td>procedure declaration or function prototype</td>
</tr>
<tr>
<td>TYLIST_TAB</td>
<td>type of each parameter in a function prototype</td>
</tr>
<tr>
<td>TY_TAB</td>
<td>types</td>
</tr>
<tr>
<td>FLD_TAB</td>
<td>field in a struct or union type</td>
</tr>
<tr>
<td>ARB_TAB</td>
<td>each entry is about a dimension of an array</td>
</tr>
<tr>
<td>TCON_TAB</td>
<td>stores integer, floating point or string constant values</td>
</tr>
<tr>
<td>BLK_TAB</td>
<td>layout of a data block</td>
</tr>
<tr>
<td>STR_TAB</td>
<td>all character defined in the user program</td>
</tr>
<tr>
<td>TCON_STR_TAB</td>
<td>character defined in the user err program</td>
</tr>
<tr>
<td>LABEL_TAB</td>
<td>an entry = a label</td>
</tr>
<tr>
<td>PREG_TAB</td>
<td>pseudo register in whirl</td>
</tr>
<tr>
<td>FILE_INFO</td>
<td>not a real symbol table. Misc info. about the compiled file</td>
</tr>
</tbody>
</table>
2.3 Inter Procedural Analysis

Software applications are typically comprised of multiple source files that are compiled separately and linked together to create an executable program. This traditional approach often leads to incomplete program information availability during compilation. To solve this problem, compilers like OpenUH perform whole program optimization. By collecting information over the entire program, the compiler can make better decisions regarding the applicability and safety of many optimization techniques.

The Inter Procedural Analysis (IPA) phase links all source files together early in the compilation process before most optimization and code generation is performed. Using intermediate representation files, OpenUH can perform inter-procedural analysis on the entire program, invoke the back-end of the compilers to optimize and generate object code, and finally invoke the standard linker to produce the final executable.

IPA is separated into two phases:

- an information gathering phase (call site information) called pre-IPA or IPL, immediately after the front end.
- the main IPA phase or optimization (IPO) phase that performs symbol analysis, constant global identification, array sections, and code layout for locality.

The analysis phase of IPA performs analysis across procedures and stores the information for later use by the IPO phase or other compiler phases such as Loop Nest
Optimizer (LNO), Global Optimizer (WOPT), and Code Generator (CG) phases. We enlist below some of the major analyses and optimizations done in IPA phase [4].

- Constructing the Call Graph - A call graph, is a multi-graph representing the static structure of a program. Each vertex represents a procedure/function of the program; each directed edge represents the relation between the caller and callee. We discuss more about the call graph in the upcoming chapters.

- Inlining - Inlining replaces calls to a function with the body of the function. Function call overhead is eliminated, and the back-end phases of the compiler are able to work on larger sections of code, potentially enabling the compiler to take advantage of other optimizations that would have been impossible when less code was available. For example, inlining may result in the formation of a loop nest that enables aggressive loop transformations. Large function and program sizes can cause higher instruction cache misses, run out of registers, use memory too frequently, or slow down later stages of the compilation process.

Due to the "whole program compilation approach", OpenUH is able to inline any function into any other function, even if they are not located in the same source file.

- IPA Cloning Analysis - This propagates information about formal parameters used as symbolic terms in array section summaries.

- Constant propagation and function cloning - Many function calls pass constants, including variable addresses, as parameters. Replacing a formal parameter with a known constant value creates opportunities for optimization. For
example, portions of a function often become unreachable and can be deleted as dead code.

- **Dead variable elimination** - Dead variable elimination removes all global variables that are never used, as well as the code that updates them, thereby speeding execution.

- **Dead function elimination** - Dead functions are functions that are never used. They can result from inlining and cloning techniques, or from continual program modification during the development process. Dead function elimination removes such functions, saving valuable space, and reducing memory and cache consumption.

- **Common padding** - Common padding improves array alignments in FORTRAN common blocks. By implementing common padding, array alignments can be improved, enabling arrays to be vectorized and accessed more efficiently and potentially reducing cache conflicts, improving performance. A similar technique can rearrange C and C++ structures.

- **Common block splitting** - Common block splitting divides a FORTRAN common block into smaller pieces, reducing data cache conflicts during program execution.

- **Procedure re-ordering** - Procedure reordering organizes functions based on their call relationship, potentially reducing instruction cache thrashing during program execution.
• Alias Analysis - Alias Analysis attempts to determine whether or not two pointers even can point to the same object in memory.

• Array Section Analysis - When summarizing a procedure, it is necessary to form the union of the sections of an array that are accessed by regions of code inside the procedure. The union of two sections may produce an approximated section since the union operation may result in a non-convex section. It is done by calling REGION * Region.Union(). This is a simple approach to compute the union of convex regions. It returns the union of two regions if they are only different on one axis and the sections on that axis can be concatenated into a single convex region. It may conservatively return NULL if it cannot find a precise union that is also convex. On the other hand, the intersection operation usually results in an exact region. The intersection of two sections is done by calling REGION* Region.Intersect() which returns the intersection of two REGIONs or returns NULL if the underestimation is BOTTOM. In mapping the caller to the callees information, it performs a reshape analysis and checks if there are aliases and global variables. It propagates information about formal parameters used as symbolic terms in array section summaries, which later will be used to trigger cloning.

2.4 The Loop Nest Optimizer

The Loop Nest Optimizer (LNO) module of OpenUH operates on High-level WHIRL. LNO is capable of solving cache-use problems and also optimize memory performance.
automatically. The compilation switch `-O3` turns on LNO.

The LNO module comprises of three phases:

- **Pre-optimization** - Consists of gathering all analysis needed for parallelization and optimization

- **Optimization** - Consists of detecting parallelizable loops (if `-apo` or `-apolistflag` is activated) and transformation for optimization

- **Post-optimization** - Consists of freeing allocated memory.

The analyses in the pre-optimization phase include data dependence analysis, array region analysis, liveness analysis, cost model and even SSA. Parallelization and transformation techniques are performed in the optimization phase. Note that if a loop is marked as *parallel*, the compiler does not perform any optimizations to avoid semantic incorrectness. Some of the main optimizers in LNO include:

- **Loop unrolling** - This is the process of reusing the loop code to include more than one iteration in a single compiler pass. Loop unrolling works by replicating the body of a loop some (machine and code dependent) number of times and scheduling the resulting code as a single basic block.

- **Hoist conditionals** - A loop may contain invariant code i.e. code which does not depend on the iterations of the loop. In such cases, the compiler hoists the code outside of the loop.
• Hoist varying lower bounds - This optimization is a variation of the above case. If the lower bounds vary, the code is hoisted outside the loop for a new set of iterations.

• Dead store elimination of arrays - This involves removing dead code related to arrays i.e. array code which does not affect the program results.

• Loop reversal - This technique reverses the order in which values are assigned to the index variable. Thus all the dependence directions are reversed. It is only legal for loops that have no loop carried dependences.

• Loop fission - Fission involves breaking down a large loop body into smaller ones to achieve better utilization of locality of reference. Its reverse action is loop fusion described below.

• Loop fusion - This is an optimization that attempts to improve performance by combining multiple loops into a single loop. The idea here is to decrease loop overhead by performing more work per loop.

• Loop tiling - This optimization scheme involves reorganizing a loop to iterate over blocks of data size which fits the cache efficiently.

• Array scalarization - This approach involves replacing subscripted variables by scalars and hence making them available for register allocation. This technique is also known as register pipelining.

• Array prefetching - Compiling irregular applications like sparse matrix vector
multiply may contain irregular array accesses, for which the array access pattern is unknown until runtime. The purpose of the software prefetch instruction is to bring data from memory into the cache before the data is required by the processor because it is usually more effective for the data to reside in the cache than the processor. The prefetch optimization in OpenUH comes from its parent Open64 compiler and is under the control of the \texttt{-LNO:prefetch} flag.

- Inter-Iteration Common Subexpression Elimination - OpenUH is capable of searching for identical expressions within loops and analyzing whether replacing them with a single variable holding the computed value is more useful.

The compiler runs the main program and then checks the return value of function FILE\_INFO\_needs\_lno(). If value returned is true, the compiler calls lno\_main() (in lno/lnodriver.cxx). LNO invokes the functions: lno\_init(), then backend\_processing() and finally lno\_fini() for termination.

### 2.4.1 Automatic Parallelization

OpenUH is capable of automatic parallelization i.e. it can analyze a sequential code and identify operations that can be executed concurrently on multiprocessor systems. The \texttt{-apo list} and \texttt{-apo keep} compiler options generate files, whose names end with .l, which list the original loops in the program along with messages telling whether or not the loops were parallelized. For loops that were not parallelized, the specific message is generated by the compiler.

APO is the synonym of Power Fortran Analyzer (PFA). In IPA, array section
analysis is done only with -ipa pfa. After IPA, auto-parallelism strategies are provided in directory be/LNO. The automatic parallelization of the compiler includes array privatization, doall and doacross parallelization, and array section analysis. Parallelization contains automatic data distribution.

Once the compiler identifies a loop with the associated dependence information, it can exploit concurrency by performing transformations such as doacross transformation and inserts necessary synchronizations. Alias analysis is performed for privatization. If doall parallelization at _parallel_depth failed, an attempt is made at parallelizing with the doacross transformation at the same level (aka parallelization with synchronization).

Some restrictions are involved in automatic parallelization. For example, the compiler often does not parallelize loops containing any of the following constructs:

- Function calls in loops
- GOTO statements in loops
- Problematic array subscripts that are indirect array references, unanalyzable or rely on hidden knowledge
- Conditionally assigned temporary nonlocal variables in loops
- Unanalyzable pointer usage in C/C++
2.5 The Global Optimizer

As the name suggests, the global optimizer is capable of invoking several optimization phases in the compiler. The inter-procedural optimization (IPO) and loop-nest optimization (LNO) rely heavily on analyses and processing offered by a global optimizer. The global scalar optimizer of OpenUH (called the WHIRL Optimizer (WOPT)) functions on mid-level (M) WHIRL. The intermediate representations include:

- Control Flow Graph (CFG) - CFG is a data structure which describes the control flow behavior of a function that is being compiled. It is a directed graph where the nodes represent the basic blocks and edges represent the transfer of control flow from one basic block to another.

- Static Single Assignment (SSA) - Usually the variables in a program tend to be assigned multiple times. SSA ensures that the program variables are assigned in exactly one location. Thus if a variable is assigned multiple times, the compiler modifies the variable name by subscripting the variable name with its version number. Variables assigned on the right-hand side of expressions are renamed to the most recent assignment. A du-chain for a variable connects a definition of that variable to all the uses it may flow to, while a ud-chain connects a use to all the definitions that may flow to it. In SSA form du-chains become explicit in the representation of a procedure thus making it trivial to figure out which definitions is are associated with which uses.

The main optimizations include:
• Partial redundancy elimination - Partial redundancy is a computation that is done more than once on some path through a flow graph, i.e., in some place along the flow graph path the computation was done and it will done again. This computation is redundant. The purpose of this optimization is to insert and delete computations in the flow graph in such a away that after the transformations each path contains no more or fewer occurrences of any such computations than before; moving computations out of loops is a subcase.

• Dead code elimination - A variable is dead if it is not used on any path from the location in the code where it was defined to the exit point of the procedure. An instruction is dead if it computes only values that are not used on any executable path leading from the instruction. The advantages of this optimizations include shrinking program size and avoiding executing irrelevant operations, which reduces its running time.

• Induction Variable Recognition (IVR) - Induction variables are variables whose successive values form an arithmetic progression over some part of the program, usually in a loop. Recognizing these induction variables and moving then out of the loop can prevent several redundant computations. Also certain computations are more expensive like multiplication operation is more expensive that add operation, IVR could help replacing expensive operations with less expensive ones.

• Value numbering based full redundancy elimination - Value numbering is a method of determining if two computations are equivalent and eliminating
one of them. It associates a symbolic value with each computation without interpreting the operation performed in such a way that computations with same symbolic value always compute the same value.

- Copy propagation - This is a process of replacing variables with their original values if the variables do not change. The optimization reduces runtime stack size and improves execution speed.

WOPT operates on code at a procedural level. It handles one program unit at a time, and its analysis and code transformation work on the entire procedure (as opposed to the more local optimization performed by the code generator (CG), and to the inter-procedural analysis handled by IPA). In addition to performing optimizing transformations to the code, WOPT computes def-use and alias information for other phases of the compiler. Depending on the level of optimization, WOPT may be invoked multiple times on the same program unit during different compiler phases. Table 2.6 represent the different WOPT phases that used in during different compiles phases.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREOPT_PHASE</td>
<td>used for PREOPT PHASE</td>
</tr>
<tr>
<td>PREOPT_LNO_PHASE</td>
<td>used for LNO phase</td>
</tr>
<tr>
<td>PREOPT_DUONLY_PHASE</td>
<td>called by LNO, but will disable optimization</td>
</tr>
<tr>
<td>MAINOPT_PHASE</td>
<td>Used when optimization level ( \ldots \leq 02 )</td>
</tr>
<tr>
<td>PREOPT_IPA0_PHASE</td>
<td>called by IPL</td>
</tr>
<tr>
<td>PREOPT_IPA1_PHASE</td>
<td>called by main IPA</td>
</tr>
</tbody>
</table>

The major difference among WOPT phases is the different level of optimizations. At -O2 and above, the MAINOPT_PHASE of WOPT is invoked just before CG.
During MAINOPT, WOPT performs its full set of optimizations and generates alias info for CG. At optimization levels higher than -O2, WOPT is also invoked for one or more PREOPT phases. During PREOPT phases WOPT generates def-use and alias info for other parts of the compiler. The various PREOPT phases do not perform the full set of WOPT optimizations. In particular, the SSA partial redundancy elimination (SSAPRE) algorithm is not run. For example, at -O3 (or -O2 -LNO), the loop nest optimizer (LNO) invokes the PREOPT.LNO_PHASE of WOPT to generate the def-use info needed to construct LNO’s dependence graph. If IPA is invoked (often as -O2 -IPA, -O3 -IPA, or -Ofast), then IPA first invokes the PREOPT.IPA1_PHASE of WOPT to generate procedure summary information IPA requires.

### 2.6 Code Generation

The final phase of the compiler is the Code Generator (CG). CG operates on a machine instructions. The intermediate representation used in the CG part of the OpenUH compiler is called Code Generator Intermediate Representation (CGIR) [31]. A CGIR representation is generated for each procedure. For each WHIRL node, one or a few machine operations (OPs) are generated and inserted into basic blocks. Each OP represents a unique target machine instruction, these definitions are described in `targinfo/<target>/isa/isa.cxx`.

The file `targinfo/<target>/isa/isa_operands.cxx` describes the operands for each operation. OPs are implemented as quads, with one or more results. Results
and operands are Temporary Names (TNs). A TN describes a register or a constant. Most of the optimizations performed in the OpenUH backend are implemented at the WHIRL levels however optimizations associated with CGIR are very target dependent.

The main phases of CG are:

1. Construct CG’s control flow graph and expand WHIRL into OP instructions.

2. Perform various loop transformations, including if-conversion, partial and full unrolling, elimination of redundant loads and stores across iterations, and the software pipeliner.

3. Carry out instruction scheduling, including general code motion (GCM).

4. General register allocation (GRA) that involves allocation of registers that cross multiple blocks.

5. Local register allocation (LRA) that involves allocation of registers localized to a single block.

6. Execute more instruction scheduling.

7. Emit assembly code and ELF/DWARF info.
2.7 Summary

The overall compilation procedure of OpenUH compiler can be summarized as follows:

OpenUH reads a source file and translates it into several levels of Intermediate Representations (IRs). These intermediate data are stored as a WHIRL Symbol Table. This symbol table contains several tables that hold information about symbols, types, initial values and procedures.

Each procedure is represented as a Program Unit (PU) which is the smallest unit of information in a call graph. A PU contains a description of the procedure and in particular a pointer to a WHIRL node which holds the intermediate representation of the body of the procedure.

The OpenUH front-end generates the first level of IR named Very High WHIRL, which is lowered during the phases of interprocedural, global and local optimizations, to eventually produce Very Low WHIRL. Very Low WHIRL is tightly bound to the underlying target and exhibits one-to-one correspondence with the target machine instructions. At this point, the IR is translated into machine code instructions to produce a Code Generator IR (CGIR), for each procedure. The last back-end transformations are applied to this representation, which include global register allocation and instruction scheduling, before the assembly is generated.

The control-flow for a procedure is built just after the WHIRL nodes have been translated into machine instructions. The smallest unit of a control flow graph is a basic block which comprise of a set of independent instructions.
Chapter 3

Dragon2011 - Architecture and Compiler Knowledge Extraction

This chapter begins by describing existing work related to Dragon2011. We formulate some key ideas which motivated us for this work. Section 2 presents the architectural modifications made to the existing framework. Section 3 provides detailed information about call graph and control flow graph knowledge extraction followed by the last section of the chapter, that gives a step-by-step walk-through on how to prepare the source program in order to invoke and use the Dragon2011 tool.
3.1 Motivation

Modern scientific programs are complex and involve application programming interfaces (APIs) to support parallelizing the code. In addition to parallelizing, these applications may be migrated to different platforms, optimized or functionalities could be extended. In all these cases, it is essential for the developer to distinctly visualize the source code. The compiler components described in Chapter 2 are capable of implementing wide spectrum of implementation strategies, intermediate-code structures, approaches to code generation and optimization, and so on. These factors have motivated researchers of the HPCTools Group at University of Houston to develop frameworks and tools which could extract exclusive source code related information from the underlying OpenUH compile. One such tool was, Dragon [38, 11, 39]. It’s first version, version 1.0, was released in 2002 followed by version 1.1 in 2005. Several works including [43, 37] have been done in Dragon to support OpenMP. OpenMP is the de facto standard for shared memory programming that can be used to program SMPs and distributed shared memory systems. Dragon’s comprehensive intra and interprocedural analysis information demonstrated promising development of an indispensable assistant for writing, analyzing and optimizing OpenMP applications.

The existing Dragon tool served as a baseline in Dragon2011’s compiler knowledge extraction techniques. We discuss the call graph and control flow graph information extraction strategies in the next section. In the list below, we present some drawbacks in the information exhibition and visualization schemes of Dragon which have motivated us for this work.
• Dragon’s GUI was developed in MOTIF/LESSTIF with X11 which restricted its usability as an application that followed specification under the X Window System on Unix and other POSIX-compliant systems only. We wanted our tool to be a portable tool which is supported on several platforms. As a solution we chose the Qt GUI development framework.

• Loading only one project at a time - When Dragon was loaded from the command line, it enabled the user to load a new project using a file selector box. Dragon accepted only input project files with extension *.d (filename.d) and a user could load only one *.d at a time. This factor restricted a user’s usability making it inconvenient to analyze multiple source codes and their corresponding graphs.

• Code browser window - Dragon provided a plain read-only text browser which displayed source code of the corresponding graphical elements, for example, clicking on a particular procedure name displayed the corresponding section of a source code containing the definition of the procedure. This information is useful however it lacked the dynamic functionality of modifying the source code. We want our tool users to modify the code and have the capability to save it. In our vision, Dragon should not be restricted to only displaying information but also serve as a code editor.

• Enhancing display of source code. Any code editor is not appreciable unless it provides syntax highlighting feature. We wanted to provide this feature to our new tool’s code browser.
• Cluttering of graphs on the user’s screen - Every time a user wished to view a call graph in Dragon, a new window popped up for each instance and similarly one window per control flow graph. When Dragon ran the NAS BT benchmark, the call graph comprised of 30 nodes thus if the user wanted to visualize all the graphs, it would result in 30 windows (for each control flow graph) + 1 (for call graph) = 31 windows on the user’s screen. This is inconvenient and in case of larger applications like GenIDLEST which comprise of more than 200 call graph elements, the scenario would be worse.

• Saving the graphs - Dragon provided print functionality where the entire graph could be printed by conserving its original dimension. Similar to Dragon, our tool would also support zoom-in/out feature. While saving the graph, we do not want to restrict saving the graph restricted to its original dimensions only but be able to save a zoomed-in or out image comprising of the specific section of the graph the user is interested in.

Additional motivational factors include:

• Our tool targets large-scale scientific applications which consist of several procedures involving multiple callee and caller relationships. The main goal of our tool is to present this information mainly in the form of graphs. Laying out several graphical items in a succinct way is a challenge and we wanted to address this by using a suitable tool or library which would provide us the appropriate algorithm. Our search led us to Graphviz library where we have used the DOT algorithm. We discuss more about this in the next section.
• We wanted to design our software as a stand-alone tool which could serve as a plug-and-play tool for the future versions of the OpenUH compiler. This involved reducing the coupling of tool’s code to the corresponding compiler files.

• Choosing Qt - Dragon2011 demanded strong GUI features which would provide faster, better and precise navigation of source-code related compiler-generated information. Since the majority of the compiler code is written in C/C++, we wanted to conform to the C++ language. Our search for an efficient and powerful GUI development framework led us to Qt. Several of our tool’s features are the result of complimentary features from Qt. The full-version of Qt SDK comes with a cross-platform integrated development environment (IDE) named Qt Creator. Qt Creator runs on Windows, Linux/X11 and Mac OS X desktop operating systems, and also supports application development on multiple desktop and mobile device platforms.

Qt’s existing sophisticated code editor, integrated UI designer classes with innumerable predefined functions, support for cross-platform qmake are some of the features which have imparted noteworthy functionalities to Dragon2011. Additionally, the active Qt development have been a considerate support in rendering ideas associated with the Dragon2011 GUI development. In the upcoming sections and next chapter, we will describe Qt-specific development details.
3.2 Architectural Modifications

Figure 3.1: Dragon version 1.1 Architecture

Figure 3.1 describes the overall architecture of Dragon version 1.1. A new module, CFG_IPL, was introduced in the OpenUH compiler infrastructure to export the control flow graph related results into the database. This new module could be enabled/disabled via compiler flags. Call graph, data dependence and array regions
information were extracted from later compiler modules and exported in similar manner. After compilation, Dragon exploited the information that had been stored in the program database, determining the mapping of results to the corresponding source code and displaying it in graphical and text forms as required. If dynamic feedback modules were enabled, the program had to be executed several times for as many runs desired by the user, prior to invoking Dragon. The program information provided by Dragon could be displayed on the screen or saved in printable formats (.vcg, .ps, or .bmp) using VCG [49].

Figure 3.2: Dragon2011 Architecture
Dragon2011 architecture is described in figure 3.2. The role of compiler-based modules evidently remain the same however the tool-browser related modules have undergone redevelopment.

When a project is loaded, the Dragon2011 tool browser reads the binary file *.d* to grab call graph information associated with the project. The graphical elements namely, the nodes and edges, are generated internally. At this point, Dragon2011 calls the Graphviz library to compute the positions of the graphical elements to be displayed on the screen. The generated *dot*(plain text graph description language) file can be dumped, if desired. Once the call graph is displayed, the corresponding control flow graphs can be viewed by reading the matching *cfg* file on the screen inside a graph dock. The Graphviz library is invoked every-time for computing the graphical element positions. Dragon2011 also displays array region analysis information by reading the *rgn* file which is a plain *txt* file. This information is displayed in the form of table.

The Graphviz [3] tool, is an open source graph visualization software which can render the *dot* file using various layout algorithms:

- **dot** - A hierarchical layout.

- **neato** - An implementation for symmetric layouts. This is a variation of multidimensional scaling.

- **fdp** - An implementation for symmetric layouts. This layout is similar to neato, but there are performance and feature differences.
• twopi - A radial layout.

• circo - A circular layout.

We use the dot layout in our implementation which is based on Sugiyama-style hierarchical layout. More background on the layout algorithm can be found here [54, 34]. Graphviz supports thirty-five output formats which can be specified with the `-Tlang` flag on the command line. By default Qt supports the following formats in table 3.1, most of which are supported by Graphviz(mentioned in bold).

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmp</td>
<td>Windows Bitmap</td>
<td>read/write</td>
</tr>
<tr>
<td>gif</td>
<td>Graphic Interchange Format</td>
<td>read</td>
</tr>
<tr>
<td>jpg, jpeg</td>
<td>Joint Photographic Experts Group</td>
<td>read/write</td>
</tr>
<tr>
<td>png</td>
<td>Portable Network Graphics</td>
<td>read/write</td>
</tr>
<tr>
<td>pbm</td>
<td>Portable Bitmap</td>
<td>read</td>
</tr>
<tr>
<td>pgm</td>
<td>Portable Graymap</td>
<td>read</td>
</tr>
<tr>
<td>ppm</td>
<td>Portable Pixmap</td>
<td>read/write</td>
</tr>
<tr>
<td>tiff</td>
<td>Tagged Image File Format</td>
<td>read/write</td>
</tr>
<tr>
<td>xbm</td>
<td>X11 Bitmap</td>
<td>read/write</td>
</tr>
<tr>
<td>xpm</td>
<td>X11 Pixmap</td>
<td>read/write</td>
</tr>
</tbody>
</table>

Qt obtains the co-ordinates from the layout algorithm and displays the graphical elements on the screen. The user has the facility to save the generated graphs in *.png format. The user can pan or zoom in/out to a particular section of the graph and print it. This module has been facilitated by Qt itself thus we did not have to include additional libraries. Note however that the user is capable of obtaining the generated dot file, generated in the intermediate steps, and visualize/save it in any of the thirty-five formats supported by Graphviz.
3.3 Compiler Extraction Modules

3.3.1 Call Graph

As mentioned in Chapter 2, the call graph extraction occurs in the Inter Procedural Analysis phases (figure 3.3) of the OpenUH compiler. The different front-end parsers perform the lexical analysis of the source code. Next, the syntactic analysis or parsing is carried out, where sequence of tokens are processed to produce the
Intermediate Representations (IR). The pre-linker gathers these intermediate data from the different source files and compilation units. After processing the IRs, the linker exports them in one or more files for backend processing. The IPL module is capable of summarizing procedure data like call site information, formal and actual parameters and global variable accesses. The global inter procedural analyzer (IPA) thus builds the complete call graph pf the program using the IPL data.

Specifically, the file ipa_cg in /osprey/ipa/main/analyze describes the data structure of the IPA_CALL_GRAPH class. The graph consists of nodes (IPA_NODE) corresponding to the procedures and edges (IPA_EDGE) corresponding to the call sites. These elements are stored in the form of vectors. In order to traverse the graph, there is a method in the IPA_CALL_GRAPH class which returns the root node of the call graph. With the root as the starting point, the call graph is traversed in pre-order fashion using the call graph iterator. The call graph structure retrieves the total size of the graph which is useful while traversing. The call graph iterator is also a useful resource to access the various IPA_NODEs. Using the IPA_NODE methods, the whole WHIRL tree can be loaded and also the corresponding symbol tables in the memory. After retrieving the node information, another iterator iterates within each node to traverse the callsite information.

Note: function **Dragon_Print()** dumps the call graph information to a binary file with the name of the output file (or target file) with the extension *.d and the corresponding *.txt file is generated by the function **Dragon_Print_Text()**.
3.3.2 Control Flow Graph

The control flow graph for a procedure is built just after the WHIRL nodes have been translated into machine instructions. Each control flow element is represented as a sequential list of basic blocks. Each basic block contains a list of operations, with a unique branch operator as the last operation. The WHIRL node from which the branch was generated is pointed to by the basic block. This WHIRL node now contains the description of the targets of the branch, which is used to create an exact control flow graph in presence of multiple target branch instructions. Internally, each control flow graph edge is represented by a list of preds and succs which represent predecessor and successors pointers of basic blocks respectively.

Dragon2011 requires an exact mapping between the control flow graph (which is generated directly from a level of IR) and the source code. For example, a contiguous sequence of statements in the source code display are directly associated with the basic block node in the control flow graph. This kind of one-to-one correspondence with the source code can only retained at VHL WHIRL. Most compiler analyses, including the construction of the control flow graph, are performed at HL WHIRL or lower levels. Thus some constructs in the source code that are directly represented in VHL WHIRL have been translated to a lower level representation before the control flow information is derived. For example, loops would appear in the control flow graph in place of array statements, leading to a source code mapping problem. The strategy used by the authors[38] was to deal with the control flow graph mapping problem separately by adding code to construct the control flow graph before VHL WHIRL is lowered. It did not affect the original Open64 analyses because the flow
graph is rebuilt in other modules as required.

In order to extract the control flow graph, new module called CFG_IPL was introduced. This module invokes the pre-optimizer when the WHIRL is still in very high level i.e. just after the front-ends. The pre-optimizer phase of the global optimizer constructs the control flow graph in the basic block mode which retrieves the file names and line numbers associated with the procedures and stores the results in the Dragon2011 database. This was implemented by developing the flow graph class, edge class and basic block class.

Since the original Open64 code did not handle the features of VHL WHIRL that are lowered, the method required to extend the existing flow graph construction code, primarily to deal with Fortran 90 features such as array statements, array sections, and the WHERE construct. There are a few limitations at present; for example, the SELECT-CASE structure is replaced by IF constructs.

Note function Print() of class FGnode in directory /osprey/be/opt, dumps the control flow graph information to a binary file with the name of the output file (or target file) with the extension *.cfg and the corresponding *.txt file is generated by the function Print_Text(). The FNode class defines all the data structures and functions associated with the nodes and edges.
3.4 Compilation Guidelines for using Dragon2011

This section gives a step-by-step walk-through of preparing your program before invoking Dragon2011. In order to use Dragon2011, the user must specify flag –enable-dragon-support while installing the OpenUH compiler. The first step of preparing a program to be analyzed by Dragon2011 is to compile it with OpenUH using a special set of flags. The user is responsible for modifying the application’s makefiles, in order to reflect the compiler and flags setting changes. Since OpenUH is based on SGI’s compiler suites, the following are the main driver commands:

- uhf90 (Fortan90/77 compiler)
- uhcc (C compiler)
- uhCC (C++ compiler)

OpenUH compiler offers several levels of optimizations. They are:

- -O : same as -O2
- -O : full optimization
- -O0 : no optimization
- -O1 : minimal optimization
- -O2 : global optimization
- -O3 : full optimization
1. Add the flags: `-ipa -O2 -dragon` for the compile (-c) and link commands as demonstrated in the example below. The `-ipa` flag will invoke the inter-procedural analysis phase and result in dumping `*.d`, that is the Dragon main project file containing call graph information. Also `*.cfg` file containing flow graph information will be generated for each source file. The user can choose any optimization flag while compiling. Note however, optimization flag, `-O3` might generate `*.dep` files, that contain data dependency information. This is on-going work and has not been integrated in the project yet. Add `-mp` flag into compile and link commands if it is an OpenMP program.

Compile commands:

```
uhf90 -c -ipa -O2 -dragon filename.f
uhcc -c -ipa -O2 -dragon filename2.c
```

Link commands:

```
uhf90 -ipa -O2 dragon filename.o filename2.o o exec
```

2. For retrieving array region analysis information, add `-IPA:array_summary:array_section -pfa` in addition to the flags declared above. After compilation, one `*.rgn` file will be generated for the project.

Compile commands:

```
uhf90 -c -IPA:array_summary:array_section -pfa -O2 -dragon filename.f
uhcc -c -IPA:array_summary:array_section -pfa -O2 -dragon filename2.c
```
Link commands:

```
ufh90 -IPA:array_summary:array_section -pfa -O2 dragon filename.o filename2.o o exec
```

In addition to static call graph and control flow graph, Dragon2011 can also display the feedback information from previous executions in the form of dynamic call graph and dynamic control flow graph. To achieve this information the following steps need to be employed:

1. Compile and link the program using uhcc/ufh90 with option

   ```
   -fb_create myfeedback -fb_type=1 -fb_phase=0
   ```

   This will generate executable files with profiling activated.

2. Run the program. A feedback file named myfeedback storing the profiling data is generated. Steps 1 and 2 can be repeated as many times as you want to merge profiling information from several executions.

3. Compile and link the program again using

   ```
   uhcc/ufh90 -fb_opt myfeedback -dragon -ipa -O2
   ```

   This will generate the dynamic call graph and dynamic control flow graph information stored in .d and .cfg file

4. Invoke Dragon2011 to load the *.d file and navigate the generated dynamic call graph/control flow graph.
3.5 Summary

In this chapter we formulated the motivational factors for this work. The main points included adapting Dragon2011 to multiple platforms and finding solutions for scalable visualization of large-scale scientific applications. To develop Dragon2011 we adopted the Qt application development framework and applied the DOT algorithm to layout the graphical elements generated and accordingly the architecture of existing Dragon tool was modified. We provided detailed description of the call graph and control flow graph extraction process where occasionally we have pointed the reader to the corresponding files in the compiler. In the final section, we have outlined the compilation steps the user must follow before invoking Dragon2011. Now that we know the theoretical structure of the elements Dragon2011 will display, the next chapter will present how to use Dragon2011 and analyze the source code information. We will also discuss the GUI-related implementation details.
Chapter 4

Dragon2011 - Design and Implementation

Section 1 of this chapter provides detailed explanation of the GUI-related source code of Dragon2011. Section 2 serves as a user manual for Dragon2011 that helps the user to understand how to accurately interpret the information that the tool provides. In Section 3, we provide the screen-shots of graphs obtained when Dragon2011 ran the NAS BT and GenIDLEST benchmarks.

4.1 Understanding Dragon2011 GUI-related classes

The Dragon2011 GUI has been implemented using the Qt application development framework. This section summarizes the flow (in figure 4.1) and functionalities (in table 4.1) of the main classes in the Dragon2011 source code. This link [19] lists all
the Qt classes, clicking on which the reader can access all the properties, public and protected functions and slots supported by the class.

![Figure 4.1: Dragon2011 - Main Classes](image)

The main function *(in main.cpp)* invokes an object of the MainWindow class. MainWindow class defines the structure of Dragon2011 main window along-with the functions for loading new project-existing source code, saving and printing the source code in the code browser, find and grep features. Loading a new project creates an object of class CallgraphWidget. CallgraphWidget class defines the layout of elements on the widget. The user can click on a button in the CallgraphWidget to view the array region analysis table. This involves invoking an object of class ArrayRegionWidget. The user can also view the control flow graph by double-clicking
<table>
<thead>
<tr>
<th>Class</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainWindow</td>
<td>Implement the Dragon2011 main window with opening / closing/ saving/ printing functionalities. The graphs are displayed in a QDockWidget implemented as QMdiArea</td>
</tr>
<tr>
<td>CodeEditor</td>
<td>Implement the code browser/editor with line numbers</td>
</tr>
<tr>
<td>Highlighter</td>
<td>Implement enhancing the code browser/editor with syntax highlighting for C/C++ programs</td>
</tr>
<tr>
<td>CallgraphWidget</td>
<td>Develop the frame in which the call graph will be displayed</td>
</tr>
<tr>
<td>ControlflowgraphWidget</td>
<td>Develop the frame in which the control flow graph will be displayed</td>
</tr>
<tr>
<td>ArrayRegionWidget</td>
<td>Develop the table structure in which the array region analysis data will be displayed</td>
</tr>
<tr>
<td>GVGraph</td>
<td>Implement the graph data structures and use the Graphviz library to layout the graph elements</td>
</tr>
<tr>
<td>DiagramItem</td>
<td>Subclassing GVNode to display the node name i.e. procedure names</td>
</tr>
<tr>
<td>MyGraphicsView</td>
<td>Subclassing QGraphicsView to panning and zooming functionalities.</td>
</tr>
</tbody>
</table>
on the procedure-list displayed on the right corner of the widget. Clicking on each instance, results in a new Controlflowgraph class object that is invoked inside the QMdiArea. Classes CallgraphWidget and ControlflowgraphWidget call following classes for specific functionalities:

- **class GVGraph** - We acknowledge Steve Dodier-Lazaro, author of this work [32], for fostering ideas for implementing this section. We use the libgraph API, that provides easy-to-use 2D graphics interface that is suitable for simple prototyping, visualization or studying graphic algorithms. The definition of the class includes defining the associated methods, beginning with the static parameters and constructor/destructor. All the attributes correspond to the Graphviz attributes. There are functions to add/remove nodes, set the appropriate font, render the graph to a file \(\text{void GVGraph::applyLayout()}\). Structures GVNode and GVEdge are defined to save the information for GVGraph’s nodes and edges respectively. Each of the nodes are then represented ad QGraphicsEllipseItem and edges as QPainterPath item.

- **class DiagramItem** - When writing Qt programs, the developer usually has to subclass Qt objects to add functionality. This is an essential concept behind creating custom widgets or collections of standard widgets. In order to add the text inside an QEllipseItem, in the form of the procedure name, we have subclassed QGraphicsObject here. Initially we used QGraphicsItem to implement this graphical element however were restricted to not using the signal/slot mechanism. We solved this by using the QGraphicsObject class to generate the graphical elements.
Note: other instances of class subclassing in this project are: class Highlighter subclassing QSyntaxHighlighter, class CodeEditor subclassing QPlainTextEditor and all our customized widgets subclassing QWidget. Another instance is MyGraphicsView subclassing QGraphicsView which is described below.

- class MyGraphicsView - This class adds zooming and panning features to the view displaying the graphs. The functionalities for the various mouse events have been implemented.

4.2 Using Dragon2011

Before invoking Dragon2011, the user has to prepare the code according to the steps described in Section 4 of Chapter 3. To run the tool the user has go to the directory where Dragon2011 is installed and type at the command prompt ./dragon2011. Note that all the graph-related screen-shots in the upcoming sub-sections are outputs obtained by running the NAS BT benchmark on Dragon2011.

4.2.1 Basic Operations

Once you invoke Dragon2011, the GUI screen shown in figure 4.2 will pop-up. To load a project, choose the *.d generated by the OpenUH compiler(example in figure 4.3).

Let us take a closer look at the main window and identify it’s main elements numbered 1-9 in the figure 4.4. We elaborate the functionalities of these numbered elements.
1. This tool bar item allows the user to load a project by choosing the corresponding *.d file. The code corresponding to file containing procedure main is displayed on the browser and the corresponding call graph is displayed in the graph dock. Once the project is loaded, the above screen will look something similar to the figure 4.5.

The region analysis data will be displayed when clicked on the button labeled View Array Region Analysis Data. The projects conforming *.rgn file will be read, an example is demonstrated in figure reffig:dragon2011-callgraph-nasbt-full.

2. This tool bar item allows the user to load any existing program to browse or modify it.
Figure 4.3: Example demonstrating loading Dragon2011 project file
**Figure 4.4:** Dragon2011 - Main Window Elements
Figure 4.5: Call graph for NAS BT benchmark

Figure 4.6: Example demonstrating array region widget invocation
3. Initiates find functionality but clearing the keyword entering field and assigning
cursor to position 1.

4. Allows user to save the current code in the code browser.

5. Allows user to print the current code in the code browser.

6. This is Dragon2011’s code browser which displays and allows editing of source
code. Note in the figure 4.7, the code browser supports syntax highlighting
feature.

Note: elements between item 6 and 7 (circled in the figure). These allow the
user to search for keywords in the code (figure 4.8). The search can be based
on an exact match or a wildcard. If ‘Grep’ is enabled, the keyword is searched
dynamically as the user types in the text field.

7. Grep Browser (figure 4.9) displays results from UNIX-like grep functionality
based on entered key string.

8. This is Qt’s Multiple Document Interface (MDI) which allows us to display
several Qt objects within the QMdiArea bounds thus preventing cluttering
of the graph elements on the user’s screen. The MDI is implemented as a
QDockWidget to allow it to float as a top-level window in the desktop. Double-
click the dock to separate it from Dragon2011 main window and double-click
it again to put it back. An example is illustrated in figure 4.10.

9. Implemented a detachable Graph Dock Widget

10. These are the menu elements of the Dragon2011’s main window.
Figure 4.7: Example demonstrating Code Browser syntax highlighting feature
** Figure 4.8: Example demonstrating find functionality **
Figure 4.9: Example demonstrating Grep Browser displaying results
Figure 4.10: Detachable Graph Dock
4.2.2 Viewing Call Graph

A Dragon2011 call graph represents the calling relationships in a program. Refer to figure 4.11 for understanding the following information. Each node represents a procedure in the program, and the directed edges represent linking a pair of nodes if the procedure corresponding to the source node may invoke the sink node’s procedure at run time. Note that the nodes may represent caller and callee in different languages. For example, a Fortran program can call a function written in C, and vice versa.

In order to give a symmetric look to the graph, we have cropped name of long procedures to a specific size. In order to know the full name, double click on the node. All the procedure names are enlisted next to the graph widget, clicking an element on the list, opens up the corresponding file in the code browser and double-clicking it, opens up the procedure’s control flow graph.

Dragon2011 displays an estimated procedure weight for each node, defined as the weighted sum of statement count, basic block count and call count within a procedure. The weight, originally designed for use by the inlining heuristic, can give an insight to the programmer on the size of procedures. The number of nodes containing OpenMP constructs is also displayed.

Using the OpenUH2011 feedback functionality, Dragon2011 displays the dynamic call graph information, which shows the cycle count and procedure frequency invoked at runtime. If the application was run several times, feedback from the different runs are collected and merged into the call graph, to show the frequencies with which procedures were invoked in different runs. This information, plus the cycle count for
Figure 4.11: Understanding call graph widget components
each procedure, can help to detect hot spots in the application, especially when there are procedures containing OpenMP. This may help the programmer decide when to inline a procedure, or where to focus manual optimization efforts on.

4.2.3 Viewing Control Flow Graph

![Control flow graph for NAS BT benchmark](image)

Figure 4.12: Control flow graph for NAS BT benchmark

Double-clicking a procedure name on the call graph procedure list displays the corresponding procedure’s control flow graph. The control flow graph represents the detailed structure of an individual subroutine or function. Refer to figure 4.12 for understanding the following information. Each node in the graph represents a basic block, a maximal segment of straight-line code with a single entry and single exit; each node is linked to all the possible successor nodes in the procedure by directed...
edges, so that each possible execution path within the procedure is represented.

The title of the Control Flow Graph window displays the name of the current procedure loaded. Table 4.2 represents all the types of nodes represented in a control flow graph along with a brief description and the color scheme in a tabular format. Similar to the call graph, the control flow graph can be saved by clicking the Print button. The format of the saved file is:

“path_to_corresponding_cfg_file”+”procedure_name”+”timestamp”.png

The time-stamp allows a user to save multiple images of the same graph. An example is demonstrated in figure 4.13.

4.2.4 Viewing Array Region Analysis data

Array Regions analysis information is displayed in a tabular structure demonstrated in figure 4.14. For each program, a procedure list is generated and displayed in the left-most column of the table. Click on each procedure name to view the corresponding array region information. A find functionality feature has been added to the table in order to search for keywords quickly.

The array regions analysis table contains five separate columns:

1. **Array Name** lists all the array variable names accessed in the selected procedure/scope. Each array was summarized based on the mode that it has been accessed with.
Figure 4.13: Example displaying control flow graph saving format
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOSTART</td>
<td>Represents DO statements (DO loop headers). There are 2 arcs connecting the DO node with the body of the DO loop and the next statement in the flow graph</td>
<td>Green</td>
</tr>
<tr>
<td>DOEND</td>
<td>Represents END DO or CONTINUE statements</td>
<td>Green</td>
</tr>
<tr>
<td>LOG-IF</td>
<td>Represents logical IF statements. If the IF statement has a True and a False branch in the flow graph, 2 arcs connect it to these branches</td>
<td>Green</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Represents entry to a procedure</td>
<td>Yellow</td>
</tr>
<tr>
<td>EXIT</td>
<td>Represents the exit to a procedure</td>
<td>Yellow</td>
</tr>
<tr>
<td>STARTREGION</td>
<td>Represents the start of a parallel region</td>
<td>Green</td>
</tr>
<tr>
<td>ENDREGION</td>
<td>Represents the end of a parallel region</td>
<td>Green</td>
</tr>
<tr>
<td>STMTS</td>
<td>Represents a set of statements with single entry and single exit, and no explicit or implicit control flow</td>
<td>Gray</td>
</tr>
<tr>
<td>STMTS/EXIT</td>
<td>Represents several statements in a basic block with the exit of the procedure</td>
<td>Gray</td>
</tr>
<tr>
<td>IO</td>
<td>Represents I/O statements including OPEN, CLOSE, READ or WRITE a file</td>
<td>Gray</td>
</tr>
<tr>
<td>REGION</td>
<td>Represents node containing a parallel region or OpenMP construct</td>
<td>Green</td>
</tr>
</tbody>
</table>
Figure 4.14: Array region analysis for NAS BT benchmark
2. **Access Mode** indicates one of the four modes of accesses: USE (array variable usage), DEF (assignment of values to array elements), FORMAL (using array variable as a formal parameter) and PASSED (an array variable passed as an actual parameter in a procedure call).

3. **Projected region range from** gives the start id of the projected region that has formed the final region.

4. **Projected region range to** gives the end id of the projected region that has formed the final region. When you add one to the difference between the value in this column and the value in column number four, it will give you the number of references in which this range of array elements has been accessed.

5. **Dimensions** column indicates the number of dimensions of the selected array variable.

### 4.3 Evaluation

We have evaluated Dragon2011 on several standalone applications including OpenMP version, and also for several languages namely - C/C++ and Fortran. The two major evaluations presented here include testing the tool on the following benchmarks:

1. Numerical Aerodynamics Simulation (NAS) - The NAS benchmarks [24] are a set of eight programs that are designed to evaluate the performance of parallel supercomputers. These benchmarks are extracted from a set of aerospace applications that mimic a class of computation in computational fluid dynamics
(CFD). Each benchmark comes in different sizes, suitable for different computer speed and memory sizes. The call graph, control flow graph and array region analysis table that has been illustrated in Section 4.2, have been obtained by testing Dragon2011 on this benchmark. The modifications made to make.def file have been illustrated in figure 4.15 following the compilation instructions provided in Section 3.4.

![Figure 4.15: NAS Makefile modifications](image)

```makefile
# This is the fortran compiler used for fortran programs
# F77 = uhf90
# This links fortran programs; usually the same as $(F77)
FLINK   = $(F77)

# These macros are passed to the linker
# F_LIB =

# These macros are passed to the compiler
# F_INC =

# Global *compile time* flags for Fortran programs
# FFLAGS = -c -IPA:array_summary:array_section -pfa -02 -dragon
# FFLAGS = -g

# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
# FLINKFLAGS = -IPA:array_summary:array_section -pfa -02 -dragon
```

**Figure 4.15:** NAS Makefile modifications
2. Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST) - GenIDLEST is a CFD simulation code written in Fortran 90 with MPI to solve the time-dependent incompressible Navier-Stokes and energy equations. We have attached the screenshots of call graph in figure 4.16. The control flow graph corresponding to one of the procedures is shown in figure 4.17, which comprises of 864 nodes. Figure 4.18 represents a zoomed-in image of figure 4.17.

4.4 Summary

This chapter starts by explaining the role of each class associated with Dragon2011’s source program. It points out the flow of invocation of the classes and describes certain programming strategies applied, for example, need for sub-classing. The main functionalities have been listed in table 4.1. The Qt development framework has played an important role in offering several functionalities to the tool.

Dragon2011 is supported on multiple platforms namely Linux and Windows. The graphical elements are laid out using the dot algorithm provided by Graphviz which has proved to be considerably scalable for large application code like GenIDLEST. The real-time search functionality for code browser and array region analysis table enhances finding keywords. The syntax highlighting feature for C/C++ renders better visualization of source code. The use of graph dock allows the user to load multiple graphs associated with same or different project, thus preventing cluttering.
Figure 4.16: Call graph for GenIDLEST benchmark
Figure 4.17: Control flow graph for GenIDLEST benchmark

Figure 4.18: Control flow graph for GenIDLEST benchmark - zoomed-in
of windows on the user’s screen. The tool additionally, offers to save the generated graphs in the desired resolution. We have tried hard to reduce redundancy in the software code especially for the graph-related data structures. Also we have tried to reduce coupling of compiler source code with GUI code and also practiced maintaining modularity of the application’s software components.

Chapter 3 explained the compiler information extraction process, and in this chapter we describe how to accurately interpret this information displayed by Dragon2011. We have added screen-shots of basic operations such as - opening a Dragon2011 project file, function of menu buttons, using find/grep feature and the graph dock. We have described what each component of a call graph and control flow graph widget operates and represents. The meaning of data in each column of the array region analysis table has been defined.
Chapter 5

Related Work

It is challenging to maintain code pertaining to current scientific applications. There are several tools which offer the user’s to understand their applications better. Based on their objective we segregate them in the following sections.

5.1 Compiler-based Tools

We highlight the special features of the following compiler-based tools which gives us several ideas for expanding Dragon2011 functionalities:

The Omni compiler offers the tlogview [5] visualization tool which helps viewing profiled OpenMP information in the ”tlog” format. It instruments an OpenMP program by inserting remote communication primitives to keep consistency of memory between nodes.
The SUIF [6] compiler infrastructure development team released tool DIDUCE and Eclipse plugins - Checklipse and KeepResident. DIDUCE targeted debugging Java programs by dynamically formulating hypothesis for strictest invariants in the beginning and gradually relaxing them as violations are detected to allow for new behavior. Checklipse used lightweight static analysis of Java sources to detect potential errors.

Cetus [56] package offers a set of utility methods which are used in common operations. They are

- **DataFlowTools**: Utility methods for detecting used or modified memory accesses
- **IRTools**: Utility methods for searching specific types of IR objects that appear in the IR tree
- **PrintTools**: Utility methods that enable pretty printing of collections and user-directed printing of verbose information
- **SymbolTools**: Utility methods related to Cetus’ symbol interface
- **Tools**: Utility methods for general purpose

The Aivi tool comes with WPP (Whole Program Parallelizer), an interprocedural parallelizing compiler which generates OpenMP programs and Hitachi SR8000 native code. As described in this paper [50], Aivi graphically displays analysis results of the input programs in the form of four windows: the call graph window, the
input window, the corresponding OpenMP program window and analysis information window. The analysis comprises of following basic properties:

1. To examine if a loop is outermost across the whole program and to find outer loops of the loop

2. To quickly find the compiler analysis information for loops and procedures in a source program or the OpenMP program generated by WPP

3. To limit the references causing data dependences interprocedurally

The KAP C/OpenMP [10] optimizer also performs an interprocedural dependence analysis (IPA) and supports several optimizations similar to our OpenUH compiler like - automatic parallel decomposition for SMP, loop optimizations, memory management optimizations, scalar optimizations, function inlining and program listings.

Polaris [26] was a parallelizing compiler developed at the University of Illinois. It mainly focused on parallelizing Fortran77 program by transforming it to one of the several parallel Fortran dialects. The six parallelization techniques were: advanced induction variable analysis, automatic inlining of subroutines, interprocedural value propagation (IPVP), array privatization, a dependence test called the Range Test, and advanced reduction analysis.

Implemented as a set of functions on the Emacs editor, the iPAt/OMP [41] is an interactive parallelizing assistance tool for OpenMP users. It was designed to offer them four types of assistance capabilities:

- Parallelism analysis: analyzing the data/control dependencies in a program
section and showing whether or not the section can be parallelized.

- Directive creation: showing the candidates of the OpenMP directives for the program section which can be parallelized.

- Program restructuring: restructuring a program section in order to enhance parallelism and parallel execution effectiveness.

- Execution time analysis: inserting calls for execution time measurement and showing the result of the measurement.

It offered text-based interaction, allowed decision-making by the user to enhance portability and extendibility, and allowed to customize its assistance capabilities according to users level of parallel programming skill.

The SUPERB [58] tool based on Vienna Fortran Compiler (VFC) [25]. SUPERB was an interactive re-structurer. It translated Fortran 77 programs into message passing Fortran for the SUPRENUM machine (a German supercomputer), the Intel iPSC, and the GENESIS machine. SUPERB performed coarse-grain parallelization for a distributed-memory multiprocessing systems (DMMP) and was also able to vectorize the resulting code for the individual nodes of the machine.

The FORESYS [22] module of the POST program development environment, was an interactive system that offered to convert programs outdated or non-standard Fortran syntax into modern Fortran77 and Fortran95 code. Foresys also had the capability to create MPI code. It was also extended to support development, verification and maintenance of OpenMP programs.
5.2 Automatic Parallelizer

Pluto [27] offered an alternative approach of automatic polyhedron source-to-source transformation framework which could optimize regular programs for parallelism and locality simultaneously. The authors implemented their framework in a tool to generate OpenMP parallel code from C program sections automatically. However, the framework was flexible to be used by any language from which polyhedra could be extracted and analyzed.

5.3 Performance Analysis Tools

One of the main objectives of Dragon2011 is to visualize information in a graphical browsable format. The following performance analysis tools offer effective GUI functionalities which can be inculcated in latter versions of Dragon2011. We discuss some of the striking features.

Tau’s [14] profile visualization tool, paraprof is capable of displaying graphics in single node/context/thread forms and hence helps the user to narrow-down the scope of visualization.

Vampir [8] offers both time-line and summary displays where the former displays (for process groups or single processes) application activities and communication along a time axis which can be zoomed and scrolled. The latter provides quantitative results for arbitrary portions of the time-lines.

In the GNU gprof [2] tool, the result of the analysis is a file which comprises of
two tables, the flat profile and the call graph. The call graph is in a tabular format and displays how much time was spent in each function. In certain cases, especially when very large number of functions are involve, we believe displaying data in a tabular format can be a more comprehensive method than graphs.

In future, we want Dragon2011 to provide an alternative view i.e. tabular view of Call Graph information. We however have yet to set up a suitable database or come up with an idea to layout this information in a suitable format.

Callgrind tool within the Valgrind [7] instrumentation framework, is a profiling tool that records the call history among functions in a program’s run as a call-graph. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls. Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behavior of an application.

In the HPCToolkit [44], we appreciate the flat view, callers view and calling context view modes in the hpcviewer’s interface. The top-down and bottom-up approach to visualize data could be a good approach to view Dragon2011’s data.

Some additional resources are Apple Shark [16], Intel VTune [12] and PTU [23], CrayPAT [1], Periscope [47] and Scalasca [35].
5.4 Code Transformation Tools

In future we want Dragon2011 to offer support for automatic code transformation/parallelization with/without user-interaction. The following tools offer some key ideas in doing so.

PARAMAT [42] comprised of a pattern recognition tool. For each pattern there existed at least one suitable parallel implementation on the target machine, parameterized in problem size and data distributions. Each implementation, in turn, had been benchmarked earlier, resulting in parameterized run time function tables which could be inspected by the run time estimator of the data distribution search algorithm. If the data distribution has been fixed, the pattern instances are replaced by the corresponding implementation. Then there only remains some residual adaptation (masks and communication insertion) for not recognized code portions, and finally, everything is translated to machine language.

TSF (Tool Set for Fortran) [45] aimed at providing the user with a tool able to accelerate most of the operations involved in Fortran code engineering. It was implemented on top of the Foresys system (mentioned earlier). It was based on a transformation script language and a case-based reasoning system. It was designed as a complementary tool to existing ones which enables to store the user's knowledge, to share it with other users and, moreover, to add new functionality to the Fortran engineering tools used by application developers.

LoopTool [48] is another source-to-source transformation tool that used a user guided approach. The tool exposed tuning parameters for external control which
offered the application programmers to choose desired performance and which also served as a back-end for empirically-driven optimization systems.

Another interesting tool is described in Orio [36]. It is an empirical performance-tuning system that takes annotated C source code as input, generates many optimized code variants of the annotated code, and empirically evaluates the performance of the generated codes, ultimately selecting the best-performing version to use for production runs. Orio also supports automated validation by comparing the numerical results of the multiple transformed versions.

Composable High-Level Loop Transformation Framework (CHiLL) [29] comprised of a compilation system that performed model-guided empirical optimization for the memory hierarchy. The compilation system comprised of (1) a transformation script which expressed to the code generator, at a high level, the sequence of transformations to be performed. (2) For each script, the engine traverses the space of parameter values, producing scripts with bound parameters that are the input of the loop transformation and code generation framework. (3) Loop transformation and code generation framework. It was built upon the Omega Library.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

Dragon2011 is an OpenUH compiler-based software tool that helps application developers or code owners who wish to understand more about their applications. The current input languages for the tool are Fortran, C/C++ and OpenMP. It is also an interactive system with a powerful GUI providing a range of information about the structure of source program in a graphical browseable form, at the level of detail desired. The tool also helps the user to efficiently navigate through these structures. 

Dragon2011 is developed using the Qt framework and uses Graphviz library to represent code structure information in a scalable graphical form. The tool also provides a rich set of visualization features. The current features of Dragon2011 include - static/dynamic call graphs with feedback information, control flow graphs
and array region analysis. The GUI features include - support for multiple platforms (Linux, Windows), syntax highlighting, scalable layout of graphical items, real-time search functionality.

6.2 Future Work

In future, from the compiler perspective, we want to explore the following enhancements to impart support for various programming models and useful analysis.

- There is on-going work in our team to support OpenMP 3.0 task dependency visualization features in Dragon2011. This paper [30] outlines the implementation of OpenMP 3.0 tasking features in OpenUH. In this dissertation [40], the author describes a strategy for the compiler to transform an OpenMP code to a collection of sequential tasks and a task graph that indicates their execution constraints. It then specifies the mapping of tasks to processors (or equivalently, to threads that are bound to processors). These work can serve as key foundation strategies for displaying task dependency information from our OpenUH compiler.

- We are also eager to add support for other programming models like Partitioned Global Space Address (PGAS) which assumes a global memory address space that is logically partitioned and is available locally to each processor. Currently there is support for the PGAS language, Coarray Fortran (CAF) [28] in the OpenUH compiler which comprises of three areas: (1) an extended front-end
which accepts the coarray syntax, (2) back-end optimization and translation, and (3) a portable runtime library. This work can be explored to add support in Dragon2011.

- We want to extend visualization of MPI-based code and support newer versions of OpenMP.

From the GUI perspective, Dragon2011 can be evolved in the following ways to serve as a high-performance visualization tool:

- 3D Rendering - In the current market, there exist tools like Avizo [57] which offer high-performance 3D visualization. Qt supports 3D visualization via Qt OpenGL module and also in alternate Qt/3D module. These can been used to render 3D graphs in future.

- Animations - We are aware of animation framework in Qt which aims to provide an easy way for creating animated user interfaces. We want to explore the animation architecture to animate our graphical objects.

- Collapsible call graph / control flow graph - Similar to the above point, we want to animate and achieve a collapsible graph in Qt which will allow the user to converge and diverge graphical elements.
Bibliography


U. Bondhugula, A. Hartono, and J. Ramanujam. A Practical Automatic Polyhe- 
dral Parallelizer and Locality Optimizer. In PLDI 08: Proceedings of the ACM 
SIGPLAN 2008 conference on Programming language design and implementation, 2008.

B. Chapman, D. Eachempati, and O. Hernandez. Experiences Developing the 
OpenUH Compiler and Runtime Infrastructure. In Cetus Users and Compiler 
Infrastructure Workshop in conjunction with PACT’11, 2011.

J. C. Chun Chen and M. Hall. CHiLL: A Framework for Composing High-
level Loop Transformations. Technical report, University of Southern California, 
2008.

L. H. Cody Addison, James LaGrone and B. Chapman. OpenMP 3.0 tasking 
implementation in OpenUH. In Open64 Workshop in Conjunction with the 

F. de Ferrière. Description of the Open64 Code Generator Intermediate Repre-


The Scalasca Performance Toolset Architecture. Concurrency and Computation: 

A. Hartono, B. Norris, and P. Sadayappan. Annotation-Based Empirical Per-
formance Tuning Using Orio. In Proceedings of the 23rd IEEE International 
Parallel & Distributed Processing Symposium, Rome, Italy, 2009. Also available 
as Preprint ANL/MCS-P1556-1008.


O. Hernandez, C. Liao, and B. M. Chapman. Dragon: A Static and Dynamic 


