# Dragon: An Open64-Based Interactive Program Analysis Tool for Large Applications

*Barbara Chapman, Oscar Hernandez, Lei Huang, Tien-hsiung Weng, Zhenying Liu, Laksono Adhianto, Yi Wen*
Department of Computer Science
University of Houston, Houston, Texas, 77204, USA
Email: dragon@cs.uh.edu, Web: http://www.cs.uh.edu/~dragon

***Abstract***- A program analysis tool can play an important role in helping users understand and improve large application codes. Dragon is a robust interactive program analysis tool based on the Open64 compiler, which is an open source C/C++/Fortran77/90 compiler for Intel Itanium systems. We designed and developed the Dragon analysis tool to support manual optimization and parallelization of large applications by exploiting the powerful analyses of the Open64 compiler. Dragon enables users to visualize and print the essential program structure of and obtain information on their large applications. Current features include the call graph, flow graph, and data dependences. On-going work extends both Open64 and Dragon by a new call graph construction algorithm and its related interprocedural analysis, global variable definition and usage analysis, and an external interface that can be used by other tools such as profilers and debuggers to share program analysis information. Future work includes supporting the creation and optimization of shared memory parallel programs written using OpenMP.

***Keywords***: Open64 compiler, callgraph, dataflow analysis, data dependences

## I. Introduction

Scientific and industrial applications are becoming increasingly large and complex. Most such applications are difficult to understand, analyze, parallelize, and optimize. A lot of effort is expended in the task of understanding a complex code structure, especially when developing parallel applications. Crucial information for any kind of program analysis, or code reengineering, includes the call graph, procedure control flow graphs and information on the variables that are involved in data dependences.

A program analysis tool can be very useful by providing this kind of information and more to aid users who need to modify or improve large codes. Traditional sequential optimizations may reorganize computation in loops or merge loops to improve cache utilization, for which data dependence information is indispensable, or they may involve multiple procedures, such as inlining functions when they are invoked inside a loop body. Parallelizing a sequential code is time-consuming and often requires considerable information on the structure of the entire code, and the data dependences in loops, something that can be difficult to obtain without tool support. Furthermore, optimization of parallel programs is challenging. For shared memory parallel codes, optimization strategies may include the privatization of scalar and array variables for improving data locality, and reducing synchronizations, all of which require considerable insight into a given code.

Dragon[1] is an interactive tool that displays program analysis results that are commonly needed to study and change sequential or parallel codes. It handles the most common programming languages and is aware of the MPI and OpenMP APIs[9] for parallel programming. We decided to use the publicly available Open64 compiler infrastructure to create this tool, since it is robust, has powerful analyses, supports multiple languages and is open source. However, this also constrains us to perform our work in such a way that we can adapt to newer versions of the compiler infrastructure. Currently, Dragon has basic features including the call graph, flow graph, data dependences and OpenMP parallel regions; it relates graphical information to the original source code wherever possible and can display both simultaneously. It is a robust tool that can visualize large graphs for large real-world applications, and it is freely available.

This paper presents the design and implementation of Dragon. Section 2 describes the Open64 compiler[8]. The following section describes Dragon, its structure, functionalities and how it was built using Open64. We briefly discuss several large applications that have been studied using this tool. Related work is outlined; and last but not least, the final section describes future work and our conclusions.

## II. Overview of Open64 Compiler

The Open64 compiler infrastructure was originally developed by Silicon Graphics Inc. and is currently maintained by Intel. Written in C++, it accepts Fortran 77/90 and C/C++, as well as a combination of any of these with OpenMP, a shared memory programming API. The system targets Intel's IA-64 processors. Open64 is a well-written compiler that performs state of the art analyses that can be exploited by tools such as Dragon. These analyses include interprocedural analysis, data flow analysis, data dependence analysis, and array region analysis. A number of different research groups already base their compiler research on this open source system.

The intermediate representation for the Open64 compiler, called WHIRL, has five different levels, starting with very high level (VHL) WHIRL, and serves as the common interface among all the front-end and back-end components.

- This work was partially supported by the DOE under contract DE-FC03-01ER25502 and by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23.

Each optimization phase is designed to work at a specific level of WHIRL. Our tool gathers information primarily from the VHL and High Level (HL) WHIRL phases, which preserve high level control flow constructs, such as do and for loops. HL WHIRL can be translated back to C and Fortran source code with only a minor loss of semantics.

The Open64 compiler basically consists of five modules as shown in Fig. 1, with multiple frontends (FE) that parse C/Fortran programs and translate them into VHL WHIRL. Additional "special-purpose" modules include automatic parallelization (APO). If interprocedural analysis is invoked, then IPL (the local part of interprocedural analysis) first gathers data flow analysis information from each procedure locally, and the information is summarized and saved in files. Then, the main IPA module generates the call graph and performs interprocedural analysis and transformations based on the call graph. Open64 next invokes the loop nest optimizer (LNO), the medium-level code optimizer (WOPT), and the code generator (CG). WOPT performs aggressive data flow analysis and optimizations based on SSA form. LNO calculates a dependence graph for all array statements inside each loop of the program, and performs loop transformations. CG creates assembly codes, which are finally transformed to binaries by AS (assembler).

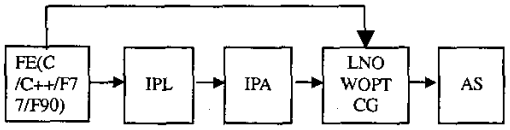


Fig. 1 The modules of the Open64 compiler

We describe IPA, WOPT and LNO in the following, because the Dragon tool uses their data structures to provide information.

**A. IPA**

In order to extract the call graph from the Open64 compiler, we require execution of both local IPA (IPL) and main IPA (pre-linker), achieved by setting the IPA flag. In this case, IPL first collects the local information of each procedure unit such as call site information, formal and actual procedure parameters, global variables accesses, and stores them in files in WHIRL format. The pre-linker reads these files and performs the call graph construction and interprocedural analyses such as global variable optimization, dead function elimination, alias analysis, cloning analysis, constant propagation, function inlining, and array region analysis for the dependence analyzer of the LNO. After that, the pre-linker saves them into files for further processing by the other module. Both nodes and edges of the call graph contain pointers that can access summary information generated by IPL.

**B. WOPT**

Depending on the compiler options selected by the users, the WOPT module may be invoked multiple times on the same program unit during different compiler phases. WOPT operates on code intraprocedurally. It lowers the VHL WHIRL for a program unit, and computes the control flow graph and basic blocks before performing data flow analysis. After computing the dominator tree, dominator frontier and control dependence set, it converts HL WHIRL to a hashed SSA form. It then performs def-use analysis, alias classification and pointer analysis, induction variable recognition and elimination, copy propagation, dead code elimination, partial redundancy elimination and more. Finally, it transforms the SSA form back to WHIRL after these analyses and optimizations.

The flow graph data structure contains the basic blocks in depth-first and post-order, the dominator tree in preorder and post-dominator tree in post-order. It also reflects the calls inside a basic block. The basic block data structure includes the type of the basic block, predecessor list, successor list, etc.

**C. LNO**

This module includes the data dependence tests. Open64 uses a hierarchical approach to look for loop level data dependences. LNO creates a dependence graph for all array statements inside each loop of the program. There is one graph per loop nest in a program unit; thus edges may only exist between an array definition and usage if they share at least one common loop. Each edge can be mapped to a dependence distance and direction vector; each edge is also linked to the WHIRL.

The data dependence test first checks for trivial cases (such as non-linear array expressions, different array dimensions, etc.). For example, if all the dimensions are too messy or contain non-linear terms, or if there is a symbolic term that varies in a loop for which we are computing a distance/direction, it returns dependence. If any dimension has two non-equal constants, the trivial test will infer independence, and otherwise continue dependence testing.

Then it performs the base test as follows. If the bases are the same and the bounds match, it continues dependence testing; but if the bases are disjoint it returns independence; if the bases overlap then it reports dependence. If these tests are insufficient, it tries to apply the GCD test; if it is still inconclusive, it solves a system of equations for exact data dependence using the Omega Test. Thus it is very powerful.

The results of data dependence analysis are used within Open64, in particular, to optimize loops. The strategy is based on a unified cost model and a model of the target cache. LNO includes well-known uni-modular loop transformations. It uses these to perform locality optimization, automatic parallelization, and OpenMP translation; heuristics integrated with software pipelining also employ LNO.

## III. Dragon Tool

We chose Open64 as an infrastructure for implementing Dragon since it contained a robust set of advanced compiler analyses satisfying our requirements[2].

Ir_tools is an Open64 utility for debugging purposes that outputs WHIRL to ascii text. We initially attempted to use this for our work. However, it does not contain any functionality for retrieving the compiler's analysis results and was thus quickly abandoned. Note that for other purposes, ir_tools can help build Open64-based tools (see Section IV).

Direct modification of the Open64 compiler was the straightforward approach, and we have developed code to extract the information needed by Dragon. It necessitated only minimal modifications to Open64. An extra module, which can be turned on or off via compiler flags, was added to Open64 to export analysis results into a program database. A user program is compiled by Open64 with our switch. Afterwards, Dragon visualizes the information stored in the database according to user needs.

Another design issue is how to display compile time analysis results in a user-friendly manner. One solution is to display the information in graphical form. For some applications, results such as the call graph, however, may be too large. For example, the POP 2.0 code[10] has a total of 428 procedures. Consequently we think that a hierarchical display model may be necessary. We display source code along with information relevant to that portion of code.

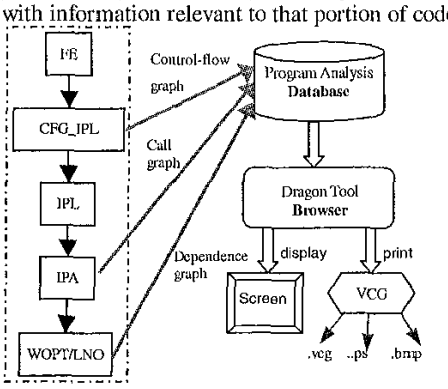

Fig. 2 The architecture of Dragon tool

Fig. 2 shows the architecture of Dragon. Dashed lines surround the Open64 modules. Useful information, such as flow graph, call graph, dependence information, is extracted from different compiler modules and exported to the program analysis database. We added one module, CFG_IPL, after FE in order to be able to map source code accurately to the control flow graph (as explained in subsection E). In this module, the control flow graph is generated and stored in the program database. IPA, WOPT and LNO are executed. Once partial compilation completes (no code is generated), Dragon exploits the information that has been gathered in the program database, mapping it with the source code and displaying it in graphical and text form as required.

The program information provided by Dragon can be displayed on the screen or saved in printable formats (.vcg, .ps, or .bmp) using VCG[12]. We describe its features in more detail below.

**A. The Call Graph**

A call graph represents calling relationships in a program. The call graph contains a node for each procedure in the program, and a directed edge linking a pair of nodes if and only if the procedure corresponding to the source node may invoke the sink node's procedure at run time. The graph can be cyclic where recursion is allowed. With Open64 and Dragon, the caller and callee may be in different languages. For instance, a Fortran program can call a function written in C, and vice versa. Our graph distinguishes the root procedure (main program), leaf procedures (that do not call any other routines), dead procedures (that are never called), and procedures containing OpenMP code via node coloring. Clicking on a node or selecting a procedure from a text list of procedure names will cause the corresponding source code to be displayed in a separate browser window. Fig. 3 shows the call graph for an OpenMP + MPI version of GenIDLEST[14].

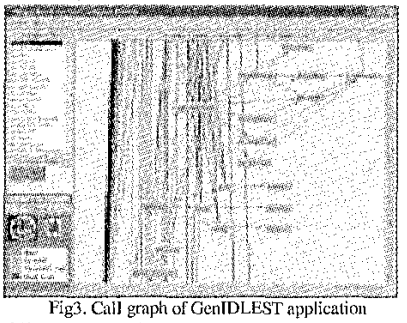
Fig3. Call graph of GenIDLEST application

Open64's call graph data structure is based on a class with a method that can retrieve a WHIRL tree or any information on a particular program unit and the corresponding symbol tables. Since the first WHIRL node corresponds to a function entry or alternate entry, we can access the source position, filename and procedure name from there. Using the call graph structure we can retrieve the total number of nodes in the graph. We determine whether the procedure contains OpenMP constructs by querying the program unit table (PU_TABLE).

Originally, the pre-linker could not retrieve the current source file name of each entry WHIRL node since the tables that contain the filenames and directory names were not updated properly. We modified the function to flush the tables purposely so as to obtain the filename.

We implemented a method that extracts the call graph in preorder fashion and stores it in the database, where each node contains information about its source code location, source file name, directory name, line number, node identification, call sites, and any OpenMP directives it has.

Open64 has difficulty handling calls with formal procedure parameters. For these it tries to use constant propagation to spread real callees, which often does not work. Weng et al.[16] developed a new algorithm to construct a precise and correct call graph. This algorithm can retrieve the exact call chains needed by many other analyses. We plan to implement this algorithm in Open64, and develop interprocedural analyses based upon it.

**B. The Control Flow Graph**

The control flow graph represents the detailed structure of an individual subroutine or function. It is the basis for dataflow analysis and many standard optimizations. Open64 constructs the control flow graph from HL WHIRL, doing so multiple times in different phases of the compiler analyses while translating a program. Dragon retrieves the control flow graph for each procedure in the IPL module. A node in the graph (Fig. 4) represents a basic block, and it is linked to all the possible successor nodes in the procedure

by directed edges, so that each possible execution path within the procedure is displayed. Nodes are colored to distinguish the entries and exits of a procedure, branches loops, and OpenMP parallel regions.

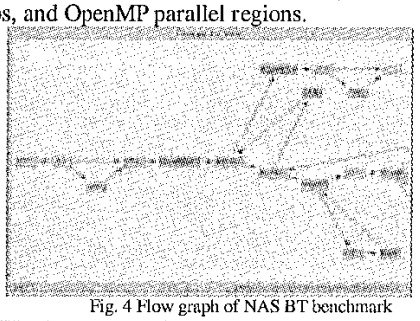

Fig. 4 Flow graph of NAS BT benchmark

We developed a flow graph class and basic block class and plugged them into the WOPT module in IPL to save the basic blocks and flow graph of each procedure, along with source code location information (directory name, file name and line number), in Dragon's database. In order to map the source code with the control flow graph accurately, the generated flow graph needs to be modified to reflect source code location, since the control flow graph construction may restructure some statements. For example, a do loop that comprises four basic blocks, INIT, END, BODY, and STEP, needs to be changed to two basic blocks, the loop condition and its body.

Dragon requires an exact mapping between the analyses and the source code. For example, a code segment should be highlighted if a basic block node in the control flow graph is selected. This one-to-one correspondence with the source code is retained in VHL WHIRL. However, the control flow graph computed in Open64 is based on HL WHIRL. That means some constructs that are directly represented in VHL WHIRL have been translated to a lower level representation. In particular, Fortran 90 is translated to Fortran 77 as part of this lowering. Without any additional work, loops would appear in the control flow graph in place of array statements, leading to a source code mapping problem.

One possible solution to this is to record the translation and mapping from VHL to HL WHIRL. This would generally help us deal with mappings between the source code and subsequent WHIRL and related analysis, but it requires non-trivial programming effort, since Open64 was not designed to keep such an exact mapping.

A simpler strategy 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. Our current system includes the CFG_IPL module that does so by invoking the pre-optimizer and storing the results in the Dragon database. It does not affect Open64 analyses because the flow graph is rebuilt in other modules as required.

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

### C. The Data Dependence Graph

Data dependence information is essential in any effort to reorganize a sequential program containing loops or to obtain a parallel counterpart. However, even though this concept is well understood by many application developers, it is notoriously difficult to detect the dependences that exist in even small regions of code. We implemented a function in Dragon to extract the data dependence graph from Loop Nest Optimizer (LNO) in Open64 compiler where each node in the data dependence graph can map to a WHIRL node. This mapping requires the traversal of WHIRL tree bottom-up until an expression or statement level node was found. From extracted data dependence graph of NAS BT benchmark code in Fig. 5, there is an edge that connects the use of rhs(n,i,j,k+1) to the definition of rhs(m,i,j,k); and therefore the Dragon will show that the loop j, i, m, n are all parallelizable, and the loop level k must be executed sequentially due to the true dependence of variable rhs.

```
do j = 1, grid_points(2)-2
  do i = 1, grid_points(1)-2
    ......
    do k=ksize-1,0,-1
      do m=1,BLOCK_SIZE
        do n=1,BLOCK_SIZE
          rhs(m,i,j,k) = rhs(m,i,j,k) - lhs(m,n,cc,k)*rhs(n,i,j,k+1)
        enddo
      enddo
    enddo
  enddo
enddo
```

Fig. 5 NAS BT benchmark example code

### D. The User Interface

To build the Dragon user interface, we used Visual Workshop from Sun Microsystems to manage the X11 /Lesstif widgets. We rely on a public domain Motif widget called XmGraph to display the call graph and flow graph. This widget has proved to be effective when dealing with large graphs, and it was very easily integrated with the rest of the interface. Each node in such a graph is a button that calls its corresponding callback function if activated.

## IV. Evaluation of Dragon Tool

Table 1: Statistics for large applications analyzed by Dragon

| Applications | No. of Procedures | No. of procedures containing OpenMP | Total number of lines |
|---|---|---|---|
| POP beta 2.0 | 428 | 31 | 46,378 |
| GenIDLEST | 214 | 80 | 48,931 |

Many midrange applications including POP, ASCI Sweep3d and UMT98, GenIDLEST and NAS OpenMP parallel benchmarks[5], have been analyzed successfully by Dragon. Both POP beta 2.0 and GenIDLEST are real-world applications containing MPI and OpenMP, as shown in Table 1. Dragon helps users easily locate procedures containing OpenMP codes, and find the parallel regions.

## V. Ongoing work

We are working on a variety of additional features, including tracking variable definitions and uses, and determining which procedures use a given global variable. SSA form is used in Open64 to perform such data flow analysis efficiently in WOPT module. Since it is not equivalent to the source code and not accurate in terms of original variables in the source code, we are using the bit-vector data flow analysis based on normal IR to retrieve the accurate variable information.

Many tools need program analysis information to perform their tasks. A common Program Database Toolkit (PDT)[6] is ideal to share analysis information with many tools. We are working toward a standard for program analysis information and interfacing Dragon with the PDT, developed by the University of Oregon. We plan to use the Open64 infrastructure to populate the analysis database so that other tools can query it.

## VI. Related Work

Foresys[2] provides a good deal of support for application improvement, but is limited to Fortran. Captools[4] transforms a sequential Fortran 77 program into MPI or OpenMP programs after asking the user questions to help the process. Others such as Source Navigator[11] accept multiple languages, multiple platforms but lack of OpenMP support. Some commercial tools including Canal can also provide program analysis but they are limited in their availability. Aivi[13] based on the WPP compiler can graphically display interprocedural analysis results of programs to help the users parallelize loops in OpenMP. *Collect* and *analyzer* in SUN Workshop are JAVA based analysis tools which allow the interpretation of profiling and performance statistics. Dragon's GUI enables a faster display than these tools.

## VII. Conclusions and Future Work

Dragon is an interactive tool that provides detailed information about a C/Fortran77/Fortran90 program that may contain OpenMP/MPI constructs. It takes advantage of Open64 analysis and capabilities. The basic information displayed in our graphical tool is general-purpose and could be employed in many situations, from analyzing legacy sequential code to helping users reconstruct parallel code. The PDT provided by Dragon enables us to collaborate with profiler and debuggers and provide wider information to users. It also shows that Open64 is a good basis for building tools such as ours.

We are also using Open64 in our research on advanced compiler teachniques, including translating to SPMD style OpenMP[6], which promises to provide good performance. We will extend data flow analysis for OpenMP programs, and compile OpenMP to macro task graphs[15] using array region analysis.

## References


[1]. Dragon tool, http://www.cs.uh.edu/~dragon

[2]. Foresys, http://www.simulog.fr/is/2fore1.htm

[3]. Hernandez, "Dragon Analysis Tool", Master Thesis. Department of Computer Science, University of Houston. December 2002.

[4]. S. Ierotheou, S. P. Johnson, M. Cross, and P. Legget, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195.

[5]. H. Jin, M. Frumkin, J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance", *NASA Technical Report*, NAS-99-011, 1999.

[6]. K. A. Lindlan, et. al. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates", Proceedings of the 2000 ACM/IEEE conference on Supercomputing., Dallas, Texas, 2000.

[7]. Z. Liu, B. M. Chapman, T.-H. Weng, O. Hernandez, Improving the Performance of OpenMP by Array Privatization. WOMPAT'2002, Workshop on OpenMP Applications and Tools. The University of Alaska Fairbanks. Fairbanks, Alaska. August 5-7, 2002.

[8]. Open64 compiler, http://open64.sourceforge.net/

[9]. The OpenMP Application Program Interface. http://www.openmp.org

[10]. Parallel Ocean Program(POP). http://www.acl.lanl.gov/climate/models/pop/

[11]. Source Navigator, http://sourcenav.sourceforge.net/

[12]. G. Sander, M. Alt, C. Ferdinand, R. Wilhelm, "CLaX, A Visualized Compiler", In *F. J. Brandenburg, ed.: Graph Drawing, Symposium on Graph Drawing GD'95, Proceedings, Lecture Notes in Computer Science 1027,* pp. 459-462, Springer Verlag, 1996

[13]. M. Satoh, Y. Aoki, K. Wada, T. Iitsuka, and S. Kikuchi, "Interprocedural Parallelizing Compiler WPP and Analysis Information Visualization too Aivi", *Second European Workshop on OpenMP ( EWOMP 2000 )*, 2000

[14]. K. Tafti. GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows. *Proceedings of the ASME Fluids Engineering Division*, FED 256, ASME-IMECE, Nov. 2001, New York.

[15]. T.-H. Weng, B. M. Chapman, "Implementing OpenMP Using Dataflow Execution Model for Data Locality and Efficient Parallel Execution". *Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-7)*, IEEE, Ft. Lauderdale, April 2002.

[16]. T.-H. Weng, Y. Wen, B. M. Chapman, "Call Graph and Side Effect Analysis in One Pass", Technical Report. Department of Computer Science, University of Houston. 2003