LANALYSIS: A PERFORMANCE ANALYSIS TOOL FOR THE IMPACT COMPILER

BY

DERek MING CHO

B.S., University of Illinois, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

ACKNOWLEDGMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support, and for giving me the chance to mature and develop new skills under his supervision. I am grateful to all members of the IMPACT research team for their advice and friendship, and I would like to offer special thanks to Rick Hank, Dan Lavery, John Gyllenhaal, Roger Bringman, and Dave Gallagher for their contributions and help with my various projects. Additionally, I would like to extend my heart-felt appreciation to all the friends who have made my college career and life so enjoyable. We will all keep in touch. Finally, I would like to thank my family: Alfred and Mona, Deidre, Brynna, and Wendy, for their love and support throughout the years; and I would like to especially dedicate this work to my sister Brynna who has endured many hardships and deserves all the happiness in the world.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The IMPACT C compiler is a state of the art optimizing research compiler being developed at the University of Illinois at Urbana-Champaign. It has gained recognition in the computer industry as a useful tool for exploring new computer architectures as well as for producing highly efficient code for existing microprocessors. The IMPACT project is constantly evolving and improving due to the hard work of many talented researchers. In the past, the complexity of the IMPACT compiler and its output have been a burden on the compiler writers themselves. Manual evaluation of code produced by the compiler can be an extremely slow and tedious process. This thesis will introduce an automated performance analysis tool, called Lanalysis, which has become an invaluable resource to researchers working with the IMPACT compiler.

1.1 Organization of the Thesis

This thesis is divided into six chapters. The remainder of this chapter will describe the motivation and the need for the compiler performance analysis tool. The second chapter will give an overview of the IMPACT C compiler project and provide an understanding of how and where the analysis tool fits into the project and its development. The third chapter will serve as a user’s manual for the performance analysis tool by describing, in detail, all of the tool’s functionality. The fourth chapter will provide a detailed account of the performance tool’s implementation. The various software components and their interactions will be thoroughly described. The fifth chapter will examine the design of the tool with respect to easy extension of its functionality. A procedure for adding analysis functions will be presented, and useful interface conventions will be shown. Finally, the sixth chapter will discuss conclusions and possible future directions for this compiler performance analysis tool.

1.2 Motivation for the Project

The IMPACT C compiler is a large software package to which contributions are made by numerous research team members. A significant percentage of these contributions and refinements comes as a result of iterative implementation and testing of new compiler features. Consequently, the development time for new code optimizations can be very slow as a result of repeated manual inspection of the compiler’s output. Detailed inspection of the compiler’s performance involves examination of the compiler’s intermediate representation of an executable program. This representation will be discussed in a later chapter, and it will become clear that hand evaluation of this code can be an extremely time-consuming task. For instance, if a compiler writer is working on an optimization dealing with large program loops, merely locating all the code associated with a particular loop can require an unreasonable amount of time. The actual analysis of the code will also be very tedious and slow without the help of a convenient evaluation tool. This thesis will present a new module to the IMPACT compiler package called Lanalysis, pronounced, “el - analysis.” Lanalysis is a tool which facilitates faster and easier analysis of the IMPACT compiler’s output.

Lanalysis has proven to be an extremely useful addition to the IMPACT compiler framework. It has greatly eased many types of code analysis for the output produced by various modules of the compiler. As a result, compiler writers are able to more quickly evaluate their programming efforts; and since implementation of code optimizations is usually an iterative process, the speed of development of new compiler features can be greatly enhanced.

2. OVERVIEW OF THE IMPACT COMPILER

IMPACT is an acronym for the Illinois Microarchitecture Project utilizing Advanced Compiler Technology [1]. The IMPACT C compiler represents the state of the art for computer code generation. This compiler system includes many optimization, profiling, simulation, and code generation modules. The compiler can produce optimized code for multiple target architectures. These architectures include AMD 29K, MIPS R3000, SPARC, HP PA-RISC, and Intel X86. In addition to targeting existing architectures, the IMPACT compiler can produce code for hypothetical "IMPACT" processor organizations. Experimental architectures can be defined through the IMPACT tools, and the code produced allows researchers to evaluate the performance of new machine designs. The following two sections will describe the IMPACT compilation process and introduce the Lcode intermediate code representation.

2.1 The IMPACT Compilation Process

During compilation using the IMPACT compiler, a "C" language source program is processed by multiple compiler modules. Additionally, the source code is transformed through several intermediate code representations. Figure 2.1 displays a block diagram of the IMPACT compiler system organization.

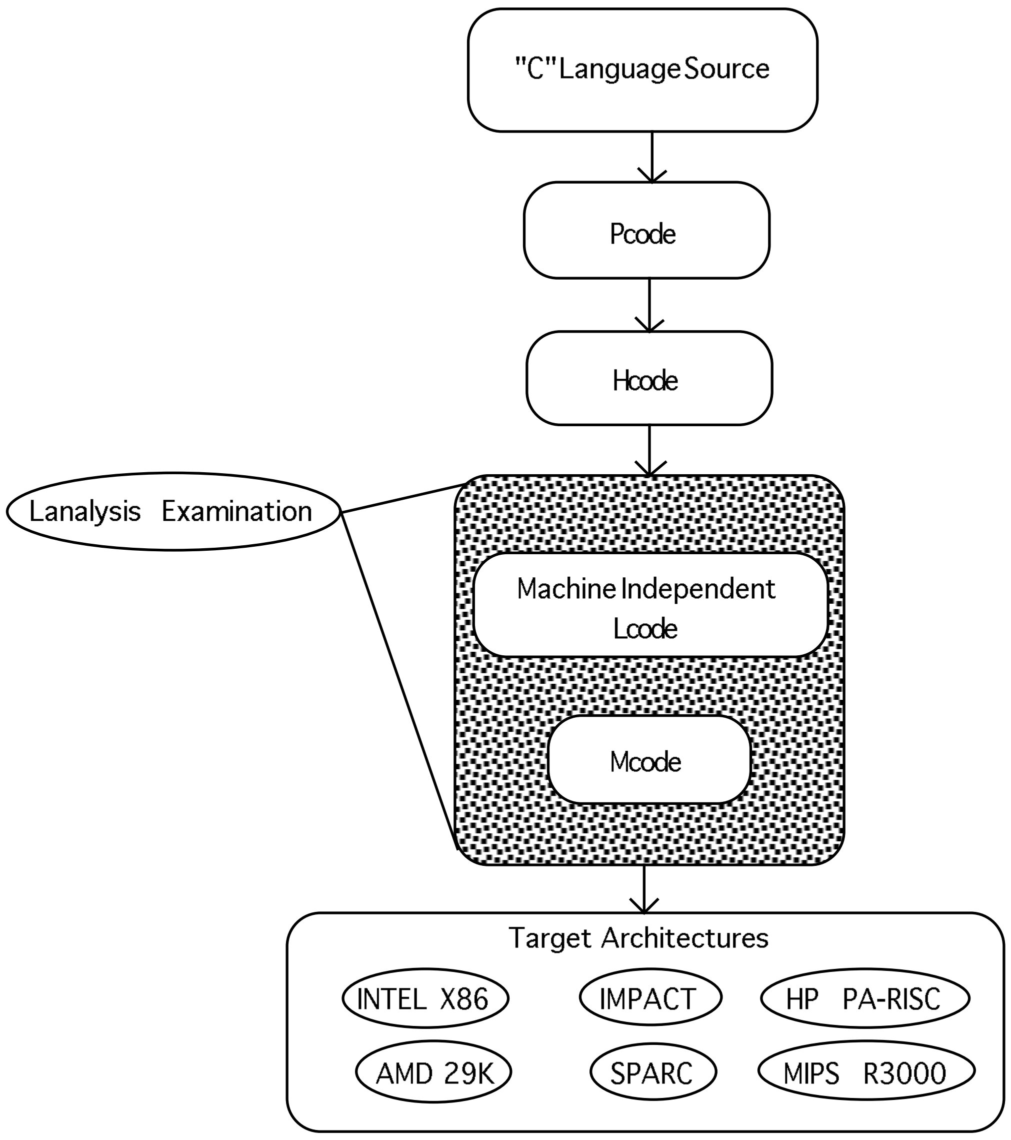


Figure 2.1: IMPACT compiler organization.

The IMPACT compilation process begins by preprocessing the source code and performing the conversion to the highest level intermediate code representation called Pcode. Pcode is a parallel “C” code representation in which loop structures are intact.

Dependence analysis [2], parallelization, loop transformations, and memory system optimizations can be performed on functions at the Pcode representation level.

Pcode functions are next translated into the Hcode format. Hcode is a flattened "C" representation containing only basic *if-then-else* and *goto* control flow constructs. Profiling probes may be inserted into the Hcode which is then reverse translated to "C" and compiled. The resulting executable will produce a profile database [3] which is merged back into the Hcode representation. In addition to program execution profiling, profile-guided code layout and function inline expansion [4] may be performed at the Hcode level.

After processing is completed at the Hcode representation level, the code is translated to the Lcode format. Lcode is a machine-independent assemblylike representation similar to many load/store RISC instruction sets. The Lcode intermediate format will be more fully described in the next section. Many optimizations can be performed by various IMPACT modules to code in the Lcode representation. These include numerous classical code optimizations [3], superblock formation optimizations [5], hyperblock formation optimizations, and instruction level parallelism optimizations. Each of these modules takes Lcode files as input and produces more highly optimized Lcode as output.

Once the machine-independent optimizations are performed, the Lcode functions are converted to machine-specific Lcode, which is named Mcode. The Mcode representation is implemented in the Lcode format, but in Mcode a one-to-one mapping exists between the target assembly language instructions and the instructions contained in the intermediate form. At the Mcode level, machine-specific peephole optimizations, register allocation, code scheduling, and final assembly language emission can be performed. After assembly language files are produced, a host assembler for the targeted machine can assemble and link the final executable.

2.2 The Lcode Intermediate Code Representation

Lcode is a text-based intermediate code format used by many modules of the IMPACT compiler [6]. Lcode operations resemble the instruction set of many RISC load/store architectures. Lcode and machine-specific Lcode, or Mcode, are the lowest level code representations used in the IMPACT compilation system. Lcode files are implemented completely in ASCII text. These files serve as the input and output of many optimization modules and provide an efficient means of communication between IMPACT compiler stages. Furthermore, Lcode and Mcode files are used as the input to the Lanalysis performance analysis tool. Figure 2.2 shows an example Lcode function.

```
(ms text)
(global _nt_compar)
(function _nt_compar 183.000000 <(FUNC (s "nt_compar")(i 136))(FILE (s "nt.c"))>)
  (cb 1 183.000000 [(flow 0 2 183.000000)])
    (op 1 define [(mac $tm_type i)] [(mac $IP i)(i 4)] <(tm (i 300))>)
    (op 2 define [(mac $tm_type i)] [(mac $IP i)(i 8)] <(tm (i 301))>)
    (op 3 define [(mac $return_type i)] [])
    (op 4 define [(mac $local i)] [(i 0)])
    (op 5 define [(mac $param i)] [(i 8)])
    (op 6 prologue [] [])
    (op 7 ld_i <F> [(r 2 i)] [(mac $IP i)(i 4)] <(tm (i 300))>)
    (op 8 ld_i <F> [(r 3 i)] [(mac $IP i)(i 8)] <(tm (i 301))>)
  (cb 2 183.000000 [(flow 1 3 183.000000)])
    (op 9 ld_i [(r 4 i)] [(r 3 i)(i 0)])
    (op 10 mov [(r 5 i)] [(r 4 i)])
    (op 11 st_i <F> [] [(mac $OP i)(i 4)(r 5 i)] <(tm (i 300))>)
    (op 12 mov [(r 6 i)] [(r 2 i)])
    (op 13 st_i <F> [] [(mac $OP i)(i 0)(r 6 i)] <(tm (i 301))>)
    (op 14 jsr [] [(l _strcmp)] <(tm (i 300)(i 301))(ret (mac $P0 i))(param_size (i 8))>)
    (op 15 mov [(r 7 i)] [(mac $P0 i)])
    (op 16 mov [(r 1 i)] [(r 7 i)])
    (op 17 mov [(mac $P0 i)] [(r 1 i)])
  (cb 3 183.000000 [])
    (op 18 epilogue [] [])
    (op 19 rts [] [] <(tr (mac $P0 i))>)
(end _nt_compar)
```

Figure 2.2: Example Lcode function.

Lcode files may hold one or more Lcode functions, and each function contains a structured list of control blocks and operations. Lcode functions, control blocks, and operations are all annotated with additional compiler information. The execution profile weights are provided for all Lcode entities, and branch target and direction statistics are listed for each control block. In addition, each Lcode construct may be annotated with extra attributes defined during various compilation stages.

When Lcode function files are read into Lanalysis or any other IMPACT module, the information contained in the file is arranged into standard memory data structures common to all IMPACT programs. These data structures may then be manipulated to perform compiler code optimizations. The Lanalysis program performs all of its performance analysis using information extracted from these data structures.

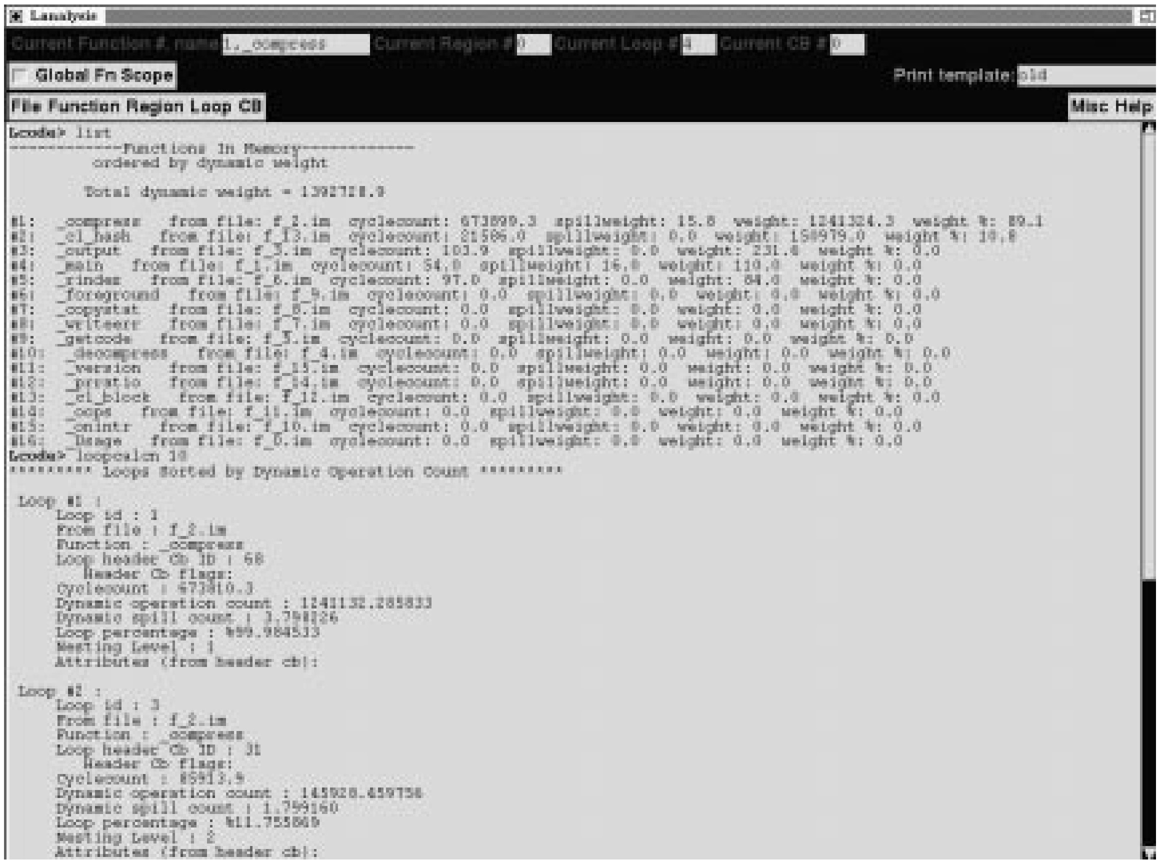
Many stages of compiler optimizations are performed after the original conversion to Lcode. The Lanalysis performance tool can analyze Lcode produced from any one of these optimization stages. Certain types of information that Lanalysis can display may not be available from Lcode taken from an early compilation module; however, execution profile statistics as well as control flow structures should be available even at the earliest Lcode representation stage.

3. FUNCTIONALITY OF THE LANALYSIS TOOL

The Lanalysis compiler performance analysis tool provides a wealth of automated and interactive analysis functions. This functionality has enabled compiler writers to quickly examine code produced by new optimizations and be presented with information that was previously time-consuming or nearly impossible to collect. The functionality includes the collection and evaluation of profiling statistics for functions, regions, loops, and individual control blocks within a compiled program; the visual presentation of program control flow graphs of functions, regions, and loops; graphical displays of dependence arcs from one operation to another within control blocks; the interactive analysis of dependence constraints among operations scheduled for a particular machine; calculation of dynamic operation counts and CPU cycle counts from programs as a whole down to individual control blocks; and many other useful data collection functions. This chapter will examine, in detail, the existing features of the Lanalysis tool. This chapter has been designed and can be viewed as the Lanalysis User’s Manual.

3.1 Basic Organization

Lanalysis has been designed as an interactive information collection and presentation tool. Once a user has loaded a set of IMPACT Lcode files into memory, many types of information can be extracted from the compiled code. The interactive nature of this analysis tool has been augmented through the implementation of a Graphical User Interface or GUI. The typical invocation of actions through pull-down menus and dialogue boxes found in mainstream applications is also available to the Lanalysis user. In addition, most commands may be executed directly by manually entering the command names at the Lanalysis user prompt. Some functions, namely those which produce graphical output, must be invoked through the pull-down menus provided by the GUI. The main Lanalysis information screen can be seen in Figure 3.1.



```
Lcode> list
-------------Functions In Memory-------------
         ordered by dynamic weight

        Total dynamic weight = 1392728.9

#1:  _compress   from file: f_2.im  cyclecount: 673899.3  spillweight: 15.8  weight: 1241324.3  weight %: 89.1
#2:  _cl_hash   from file: f_13.im  cyclecount: 21506.0  spillweight: 0.0  weight: 150979.0  weight %: 10.8
#3:  _output   from file: f_3.im  cyclecount: 103.9  spillweight: 0.0  weight: 231.4  weight %: 0.0
#4:  _main   from file: f_1.im  cyclecount: 54.0  spillweight: 16.0  weight: 110.0  weight %: 0.0
#5:  _rindex   from file: f_6.im  cyclecount: 97.0  spillweight: 0.0  weight: 84.0  weight %: 0.0
#6:  _foreground   from file: f_9.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#7:  _copystat   from file: f_8.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#8:  _writeerr   from file: f_7.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#9:  _getcode   from file: f_5.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#10:  _decompress   from file: f_4.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#11:  _version   from file: f_15.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#12:  _prratio   from file: f_14.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#13:  _cl_block   from file: f_12.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#14:  _oops   from file: f_11.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#15:  _onintr   from file: f_10.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
#16:  _Usage   from file: f_0.im  cyclecount: 0.0  spillweight: 0.0  weight: 0.0  weight %: 0.0
Lcode> loopcalcn 10
********* Loops Sorted by Dynamic Operation Count *********

 Loop #1 :
     Loop id : 1
     From file : f_2.im
     Function : _compress
     Loop header Cb ID : 68
        Header Cb flags:
     Cyclecount : 673810.3
     Dynamic operation count : 1241132.285833
     Dynamic spill count : 3.798126
     Loop percentage : %99.984533
     Nesting Level : 1
     Attributes (from header cb):

 Loop #2 :
     Loop id : 3
     From file : f_2.im
     Function : _compress
     Loop header Cb ID : 31
        Header Cb flags:
     Cyclecount : 85913.9
     Dynamic operation count : 145928.459738
     Dynamic spill count : 1.799160
     Loop percentage : %11.755869
     Nesting Level : 2
     Attributes (from header cb):
```

Figure 3.1: Lanalysis main screen.

In addition to the use of Lanalysis through the graphical user interface, there are two alternative methods of using the tool. One is to use Lanalysis interactively in a text-only mode, which is quite similar to the normal use of the program. The other method is to use Lanalysis in a non-interactive manner in which a script of Lanalysis commands is fed to the tool and is executed automatically. The output may be written to a single or multiple text files. Instructions for the various methods of using the Lanalysis tool will be presented in Section 3.4.

Another notable feature of the Lanalysis tool is that it can make direct use of another IMPACT module called Lpretty, which is a program that can "pretty print" IMPACT Lcode files. The way in which a standard Lcode file is printed is based upon a template that can be defined by the individual user. There are numerous ways in which Lpretty can format and accentuate particular pieces of information contained in an Lcode file, and several useful print templates have been defined. From within the Lanalysis tool, an existing Lpretty template may be selected. If a user chooses to merely print an Lcode function or control block from within Lanalysis, the tool will automatically use Lpretty and the template chosen by the user to format the Lcode text. This can be very useful; especially to IMPACT users who are accustomed to viewing Lcode using their own personal Lpretty template.

An issue that Lanalysis users should be aware of is that many of the tool's analysis functions are only useful if the Lcode is annotated with particular pieces of information. This may require that the Lcode being examined has passed through specific modules of the IMPACT compiler. For example, many Lanalysis functions assume that the code contains execution profile information. This would require that the profiling steps in the IMPACT compile path have been performed. Almost every Lanalysis command assumes the existence of execution profile information. The descriptions provided for each Lanalysis function will include any assumptions of additional information availability made by the tool.

3.2 Getting Started

In order to run Lanalysis successfully, there are several things that must be configured correctly. First of all, Lanalysis and its supporting software have been compiled for both the Sun and Hewlett-Packard workstations, so the user will have to run the executable on one of these machines. Also, it is recommended that Lanalysis be displayed on a color monitor since much of the graphical output is color coded to convey certain types of information. Second, the user should insure that the needed software is available and visible from the present working directory. Finally, the input files and IMPACT standard parameter file must be produced and configured correctly.

The software needed to run Lanalysis consists of the following: Lanalysis, wish, lanalysis, Ldag, and Lpretty. *Lanalysis* is the main module of the package; it performs all the major analysis and calculation functions provided by the tool. The *wish* program is an interpreter for scripts written in the Tk/Tcl programming language. This is used to interpret and execute commands contained in *lanalysis,* which is a Tk/Tcl script that implements the graphical user interface and produces all of the tool's graphical output. *Ldag* is a module which computes the layout of directed acyclic graphs. It has actually been enhanced in order to handle cyclic graphs, so the name of the module is slightly misleading. This program is used by Lanalysis to produce various types of directed graphs. *Lpretty* was described previously and is called by Lanalysis to format raw Lcode function text for display. These modules are more fully described in Chapter 4.

Lanalysis takes IMPACT Lcode files as input. These files are produced as a result of partial IMPACT compilation of a "C" program. The options specified in the parameter file (STD_PARMS) used during this compilation should also be set in the STD_PARMS file used by Lanalysis. In particular, it is important that the processor architecture and model specifications, as well as the IMPACT machine description file [7], are set equivalently in both standard parameter files. Usually, the same STD_PARMS file is used during compilation and examination with the Lanalysis tool. Additionally, Lanalysis requires that a specific Lanalysis section be present in the STD_PARMS file. The lines shown in Figure 3.2 should be included in any STD_PARMS file used with Lanalysis.

```
(Lanalysis $rel_parms$/LANALYSIS_DEFAULTS
end)
```

Figure 3.2: Required Lanalysis STD_PARMS section.

The default Lanalysis settings are shown in Figure 3.3. The default parameter values may be changed by adding specific parameter definitions to the Lanalysis section of the STD_PARMS file being used. This method of parameter assignment is common to all IMPACT compiler modules.

```
examine_zero_cycle_dependences = no;
use_cyclic_dependence_graphs = no;
non_interactive_mode = no;
template = old;
```

Figure 3.3: Default STD_PARMS settings.

Once the mentioned environment requirements are met, the user is ready to begin work with Lanalysis. The following section will list and describe all of the existing functionality of the tool. Normal use of the of the performance analysis tool is invoked by entering “lanalysis” at the UNIX prompt.

3.3 Current Functionality of the Lanalysis Tool

As mentioned earlier in this chapter, there are several methods of using the Lanalysis tool. The complete functionality of Lanalysis is not available to every method of the tool’s use. This section will be divided into two subsections. The first will document all features of the tool which provide only text output. These features are available to every method of using Lanalysis. The name of each command will be given in bold print and will be followed by its description. The second subsection will describe the analysis functions which provide graphical output. These functions are available only through the use of Lanalysis under the graphical user interface.

3.3.1 Functions with text output only

load - This command loads Lcode files into the Lanalysis tool. It takes one argument which is the name of an Lcode file. This name can contain the usual UNIX wildcard characters which allow the user to load multiple files with a single command. Files must be loaded before any analysis can be performed.

startfile - This command is used to start an output file to which all text output will be written. This command takes one argument which is the name of the output file.

endfile - This command will close the output file started with **startfile**. The text output will no longer be written to the file.

reset - This command will restart the Lanalysis tool. All functions in memory as well as information collected will be discarded.

quit - This statement will end an Lanalysis session by exiting the tool.

settemplate - This command will set the template to be used by Lpretty for displaying Lcode text. The template name must be defined in the "style file" being used by Lpretty. See the on-line Lpretty documentation for more information on this. The default template is "old".

? - Entering a question mark will display a list of all commands available at the Lanalysis command line.

list - This command will list all functions that have been loaded into memory. The order of the list is determined by each function's dynamic operation count (or dynamic weight). Functions with the highest dynamic operation count will be at the top of the list. Other information included in this list are the sum of the dynamic weights of all functions in memory, the dynamic weight percentage of individual functions with respect to the total sum, the estimated CPU cycle count of each function, the number of dynamic register spill code operations in each function, the name of each function, and the name of the Lcode file from which each function was loaded. The availability of some of these statistics is dependent upon the IMPACT compile path. Profile information must exist. Register allocation must be performed for spill code information. Code must be scheduled for a particular machine for CPU cycle counts to exist.

listbycyc - This command presents the same information as the **list** command; however, the list is ordered by the CPU cycle counts of each function. Also, the total CPU cycle count and individual function percentages are based on cycle counts rather than dynamic operation counts. It can be useful to compare these results to results from the **list** command. Code must be scheduled for CPU cycle counts to exist.

formlistfunc - This produces exactly the same information as the **list** command, but the output is formatted differently, providing more explicit statistic identification. Each piece of information is listed on a separate line, and complete labels are given.

formlistfunccyc - This produces the same information as the **listbycyc** command, but the output is formatted differently, providing more explicit statistic identification. Each piece of information is listed on a separate line, and complete labels are given.

listfuncweights - This command lists the *static* weights of each function. These are the number of times each function was called for the particular input profiled. The order of this list is determined by the dynamic operation count of each function.

listfuncspills - This function lists the number of register spill code operations in each function as well as the percentage of each function accounted for by spill code. The order of the list is determined by the dynamic operation count of each function. Register allocation must have been performed.

totalspillcount - This command returns the total number of spill code operations from all functions in memory, and is useful for evaluating register pressure for a program as a whole. Register allocation needed.

setcurrent - This command is used to choose the function in memory to be examined by subsequent commands. It takes one argument which is the function’s i.d. number. This number is determined by the function’s placement in the dynamic operation count list. Use the **list** command to determine a function’s i.d. number.

setfnbyname - This function is equivalent to the **setcurrent** command, but the command’s argument is the actual name of the function being selected rather than its i.d. number.

current - This returns the name and number of the current function being analyzed.

print - This command will display the Lcode text of the current function. The text will be formatted according to the Lpretty template selected.

operweights - This function will display the profile weights of each operation in the current function. The profile weights of each control block can also be viewed with this command.

totaldynopcount - This statement will return the total dynamic operation count of all functions loaded in memory. This information is also available from the **list** command, but this command is useful if the entire function list is not required.

totalcyclecount - This command will return the total CPU cycle count of all functions loaded in memory. This information is also available from the **listbycyc** command, but this command is useful if the entire function list is not required. Code must be scheduled.

setcurrentregion - This command is used to set the current region, within the current function, that will be examined by subsequent commands. It takes one argument which is the i.d. number of the region being selected. Note that a current function must be selected in order for this command to have any meaning. Also, it is necessary to have function regions formed during the IMPACT compile path.

regioncalcn - This command will return a list of regions from within the current function. The command takes one argument which is the number of regions to be returned. The list will be ordered by the number of dynamic operations in each region. Therefore, providing an arguments of "5" would return the top five regions. Information contained in the list will include the i.d. number of each region, name of the function from which the region came, the dynamic weight of each region, and the percentage of the dynamic weight with respect to the weight of the entire current function. Region formation required.

regioncalcalln - This command will return a list of regions similar to that produced by the **regioncalcn** function, but this will be a list of regions from *all* functions in memory. It takes one argument which is the number of regions to be returned. The region weight percentage will be given with respect to the weight of all functions in memory. Region formation required.

setcurrentloop - This command is used to set the current loop, from within the current function, that will be examined by subsequent commands. It takes one argument which is the i.d. number of the loop being selected. Note that a current function must be previously selected for this command to have any meaning.

loopcalcn - This command will return a list of loops from within the current function. The command takes one argument which is the number of loops to be returned. The list will be ordered by the number of dynamic operations contained in each loop. Information contained in this list will include the i.d. number of each loop, the name of the function in which each loop is contained, the file from which that function was loaded, the i.d. of the loop header control block, flags set for the loop header control block, attribute values from the loop header control block, the dynamic operation count of each loop, the dynamic operation count percentage with respect to the current function for each loop, the dynamic register spill code count from each loop, the CPU cycle count for each loop, and the nesting level of the individual loops.

loopcalcalln - This command will return a list of loops similar to that produced by the **loopcalcn** function, but this will be a list of loops from *all* functions in memory. It takes one argument which is the number of loops to be returned. The loop weight percentage will be given with respect to the weight of all functions in memory.

loopcyccalcn - This command returns a list with the same information returned by **loopcalcn**, but the list's order is determined by each loop's CPU cycle count as opposed to dynamic operation count. The command takes one argument which is the number of loops to be returned. Also, the cycle count percentage will be given in place of the dynamic weight percentage. Code must be scheduled.

loopcyccalcalln - This command will return a list of loops similar to that produced by the **loopcyccalcn** function, but this will be a list of loops from *all* functions in memory. It takes one argument which is the number of loops to be returned. The loop cycle count percentage will be given with respect to the total cycle count of all functions in memory.

printlooptabn - This command will return the same information as the **loopcalcn** function, but the output will consist of pieces of information separated by TAB characters. This is useful if the output is written to a file that will be loaded into some sort of spreadsheet program. The command takes one argument which is the number of loops to be returned.

printlooptaballn - This command returns a list of loops similar to the TAB formatted list returned by **printlooptabn**, but this will be a list of loops from *all* functions in memory. The command takes one argument which is the number of loops to be returned. The loop percentages will be given with respect to all functions loaded.

setcurrentcb - This command is used to set the current control block, from within the current function, that will be examined by subsequent commands. It takes one argument which is the i.d. number of the control block being selected. Note that a current function must be previously selected for this command to have any meaning.

printcb - This command will display the Lcode text for the current control block. The text will be formatted according to the current Lpretty template selected. Note that a current control block must be selected for this command to produce any output.

cbcyclecount - This function will return the calculated CPU cycle count of the current control block. The code must be scheduled for a machine for this command to be meaningful.

docfrequentn - This command will return a list of control blocks from within the current function. The command takes one argument which is the number of control blocks to be returned. The list will be ordered by the dynamic operation count of each control block. Information contained in this list will include the i.d. number of each control block, the function in which the control block is located, the dynamic operation count of each control block, the dynamic operation count percentage of each control block with respect to the function containing the block, the CPU cycle count of each control block, the number of dynamic register spill code operations in each control block, flags set for each control block, and attribute values associated with each control block.

docfrequentalln - ** This command will return a list of control blocks similar to that produced by the **docfrequentn function, but this will be a list of control blocks from *all* functions in memory. It takes one argument which is the number of blocks to be returned. The control block dynamic weight percentage will be given with respect to the weight of all functions in memory.

cycfrequentn - This command returns a list of control blocks with the same information returned by **docfrequentn**, but the list's order is determined by each control block's CPU cycle count as opposed to dynamic operation count. The command takes one argument which is the number of control blocks to be returned. Also, the cycle count percentage will be given in place of the dynamic weight percentage. Code must be scheduled.

cycfrequentalln - This command will return a list of control blocks similar to that produced by the **cycfrequentn** function, but this will be a list of control blocks from *all* functions in memory. It takes one argument which is the number of blocks to be returned. The control block CPU cycle count percentage will be given with respect to the cycle count of all functions in memory.

mostfrequentn - This command returns a list of control blocks with much of the same information returned by **docfrequentn**. The difference is that raw profile weight of the control blocks is used in place of the dynamic operation count; the latter is the sum of the profile weights of all operations contained in the block. The list is ordered by the profile weights of the control blocks. The command takes one argument which is the number of control blocks to be returned. The weight percentage is given with respect to the sum of the profile weights of all control blocks in the function.

mostfrequentalln - This command will return a list of control blocks similar to that produced by the **mostfrequentn** function, but this will be a list of control blocks from *all* functions in memory. It takes one argument which is the number of blocks to be returned. The control block weight percentage will be given with respect to the sum of the profile weights of all control blocks from all functions in memory.

dependenceinfo - This command will print the dependence information for the current control block using the standard IMPACT dependence printing functions. The Lcode text for each operation in the control block will be printed along with a list of the operation's input and output dependences and level in the dependence graph. A current control block must be selected for this command to be meaningful.

utilization - This command will return a text-based representation of the control block's scheduled operations. It will also return the percentage of issue slots filled for the particular machine as a measure of the resource utilization of the code. The code must be scheduled for this command to return meaningful information.

In addition to these text-based commands, the following are also available: **loopcalc, loopcalcall, printlooptab, printlooptaball, loopcyccalc, loopcyccalcall, mostfrequent, mostfrequentall, docfrequent, docfrequentall, cycfrequent,** and **cycfrequentall.** These commands return exactly the same information as the similarly named commands ending with "n," but these commands do not let the user specify the number of items to include in the information list. These commands can return extremely long lists, so they should be used only if the complete information lists are needed.

3.3.2 Functions with graphical output

This subsection will describe the Lanalysis functions which produce graphical output and which may only be invoked through the menus provided by the graphical user interface. These functions fall into three categories: control flow graphs, dependence arc graphs, and analysis of dependence constraints for a given control block schedule. The interactive functionality of these tools will be presented.

The first group of graphical Lanalysis functions deals with the production of program control flow graphs. These control flow graphs are generated and displayed such that each node of the directed graph corresponds to an individual basic block of the function's code. Each node will be labeled with the i.d. number of the basic block it represents and will be connected to other nodes by directed arcs which indicate the execution path of the function. Forward branching arcs are displayed in cyan, and backward branching arcs are displayed in red.

Control flow graphs may be produced for a variety of program constructs. Graphs may be generated for functions as a whole, regions of functions, and individual loops within functions. The commands used to invoke these functions are located inside the “Function”, “Region”, and “Loop” pull-down menus. To produce a control graph of a function, that function must be previously selected as “current” by the user. When graphing a region, a current function and current region within that function must be selected. Similarly, when graphing a loop, a current function and current loop within the function must be selected. A typical control flow graph can be seen in Figure 3.4.

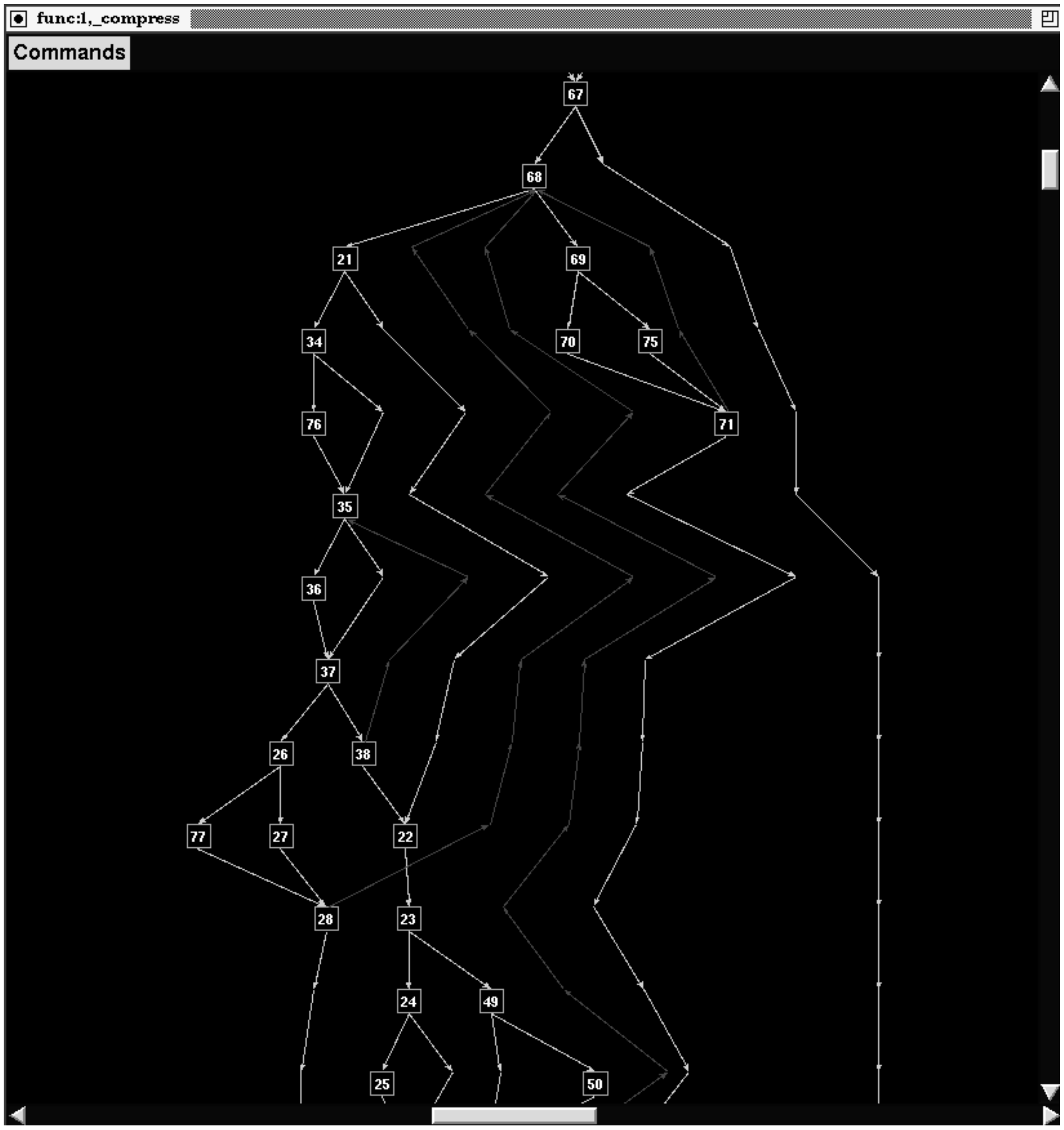
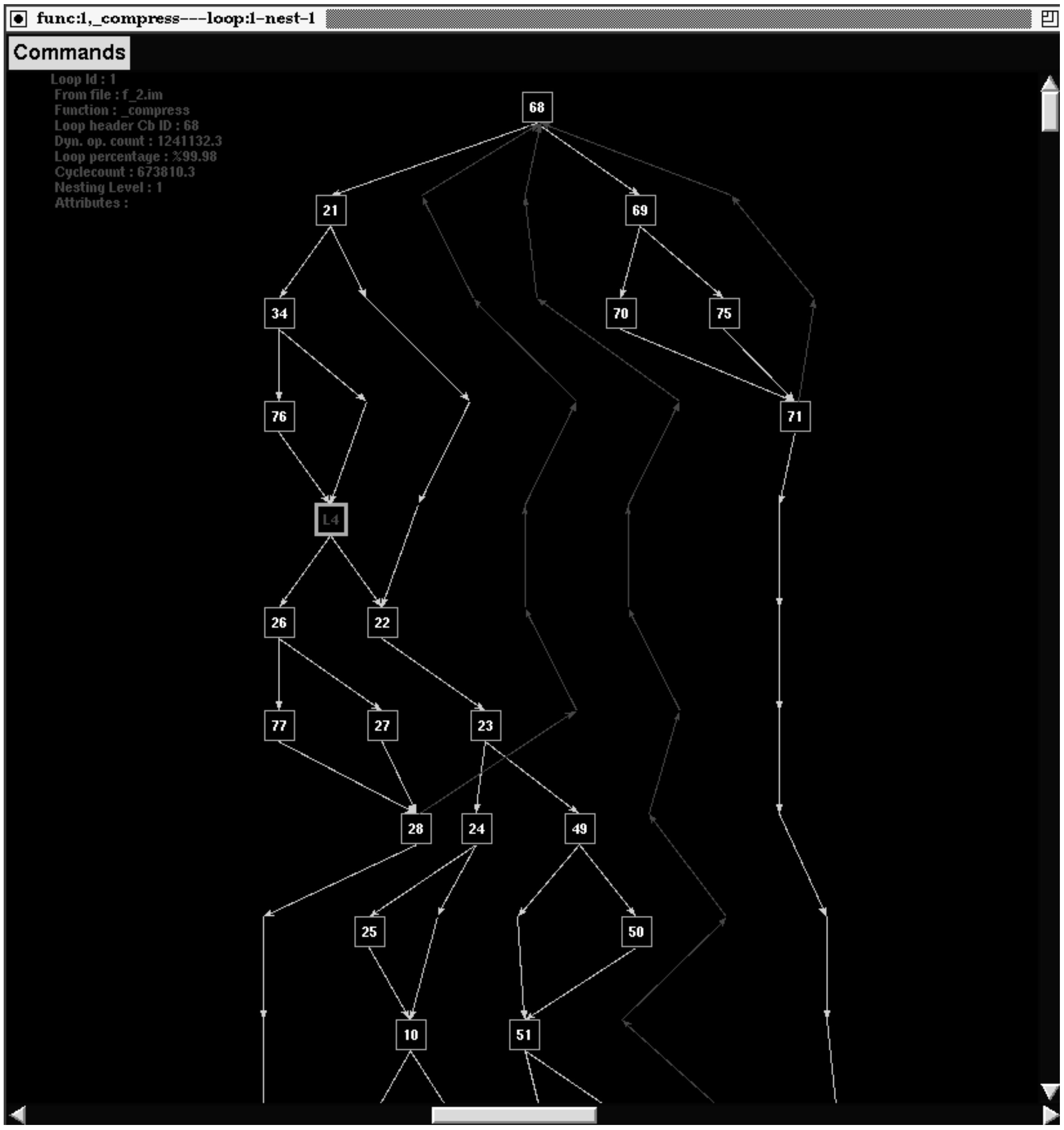


Figure 3.4: Control flow graph.

Control flow graphs for both functions and loops may be displayed to a specified loop nesting level. This allows users to reduce the size of the graphs by a significant amount which can make their viewing more manageable. If this option is selected, the user will be prompted for the loop nesting level to which individual control blocks will be displayed. The default level is "1" which corresponds to code at the outermost nesting level. Loops which are nested beyond the value specified will be condensed and displayed as a single highlighted node in the graph. An example of a function graphed to one nesting level can be viewed in Figure 3.5. Note that the node labeled "L4" represents a more deeply nested program loop.

Once a control flow graph is produced for a particular program structure, several things may be done with the graph. First of all, the graph can be adjusted visually for optimal viewing. The window in which the graph is displayed may be resized, the user can zoom the image both in and out, and the graph's vertical scale can be independently adjusted to the viewer's taste. Second, the user may produce both Postscript and Color Postscript files from the control flow graph which may then be sent to a printer. The Postscript files produced will contain whatever portion of the graph is visible at the time the files are generated. The resizing and printing options are available through the pull-down menu in the graph window. Lastly, additional information may be obtained through interactive examination of the control flow graph. Individual nodes in the graph are produced as "buttons" which may be activated using the mouse. Selecting a node representing a control block with the left mouse button will create a resizable window containing the Lcode text for that control block. This text will be formatted according to the current Lpretty template. The user may view profile information, branch direction statistics, as well as the control block's code through the use of this feature. Selecting a node representing a collapsed loop of a greater nesting level will produce a control graph of that loop to the next nesting level. If such a node is selected with the middle mouse button, the user will be prompted for the number of nesting levels to include. All the functionality available to the original graph will also exist for the new control flow graph.



```
Loop Id : 1
 From file : f_2.im
 Function : _compress
 Loop header Cb ID : 68
 Dyn. op. count : 1241132.3
 Loop percentage : %99.98
 Cyclecount : 673810.3
 Nesting Level : 1
 Attributes :
```

Figure 3.5: Control flow graph one nesting level deep.

The next type of Lanalysis functionality that produces graphical output deals with the creation of dependence arc graphs, which may be produced for code contained in a single control block. Lanalysis can create directed graphs for control blocks in which each node of the graph represents a single operation, and arcs between nodes represent dependences between operations. Nodes in the graph will be labeled with the i.d. number of the particular operation it represents.

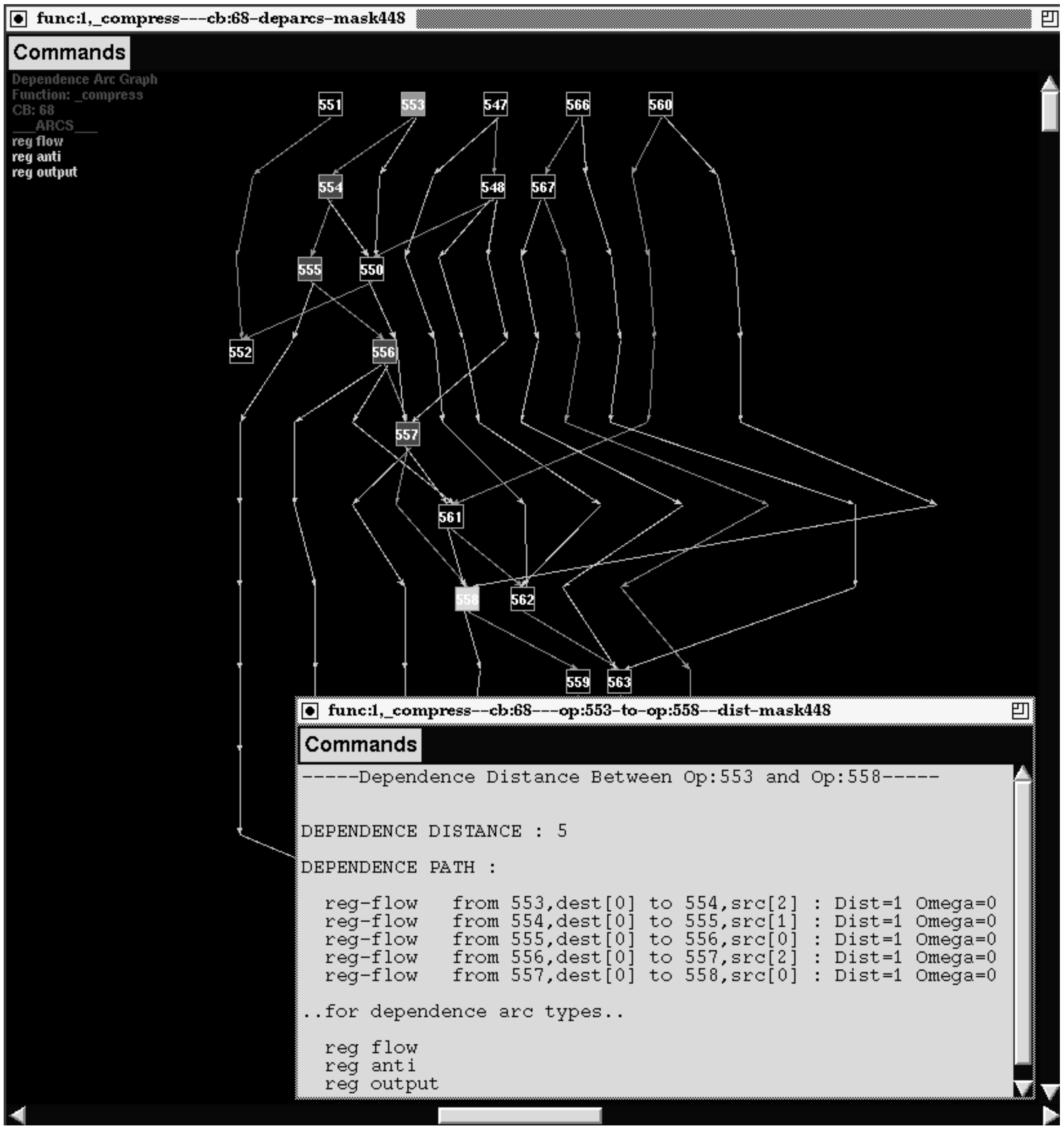
Dependence arc graphs can be used to display information for a variety of dependence types. In order to display a dependence arc graph, a current function and a current control block within that function must be selected. Next, "Dependence Arc Graph" should be selected from the control block pull-down menu. At this point, a window will appear which allows the user to select the types of code dependences to be displayed. The choices include: register flow, register anti, register output, memory flow, memory anti, memory output, control, and synchronization dependences. Once the dependence types have been chosen, a window containing the graph will appear. The arcs in the graph will be color coded to identify the type of dependence arc they represent. This window may be resized, and the view may be adjusted in the same fashion as for control flow graphs. The same Postscript printing functions are available.

Several interactive analysis functions can be performed on these dependence arc graphs. The nodes representing operations are "buttons" that can be activated with the mouse. Selecting an operation using the left button of the mouse will produce a resizable window containing the Lcode text for that operation as well as a list of all incoming and outgoing dependence arcs. The dependence types, source and destinations, distances, and iteration distances will be displayed for each dependence arc in the list. Note that iteration distance is only meaningful if cyclic dependence graphs are produced; this will be discussed in this section. Through the use of the center mouse button, the maximum dependence height and path between two operations can be found. This is accomplished by pressing the center mouse button on an operation, and then pressing the center mouse button on a second operation. The first operation will be highlighted in blue, and the second will be colored grey. All operations on the maximum height dependence path between the two operations will be highlighted in red. Additionally, a window which contains a description of this dependence chain will be produced. It should be noted that only dependence arcs of the types selected for the graph will be examined in the dependence path calculations. Pressing the right mouse button will clear all highlighted operations. An example of a dependence arc graph can be seen in Figure 3.6.

Both cyclic and acyclic dependence analysis may be performed for code contained in a control block. The default analysis type is non-cyclic. Cyclic dependence analysis can be useful for code on which software pipelining optimizations have been performed. There is a parameter, “use_cyclic_dependence_graphs”, which can be set in the Lanalysis section of the STD_PARMS file, that controls the type of dependence analysis performed. This parameter may be set interactively using the “change parameters” option under the “miscellaneous” pull-down menu.

The third graphical Lanalysis tool deals with the analysis of dependence constraints for a given control block schedule. A graphical representation of scheduled operations may be produced for code contained in a single control block. A grid with as many slots as the issue width of the targeted machine will be filled with operations, reflecting the utilization of issue slots for each cycle of program execution. Data dependence constraints between operations may be examined for insight into scheduling choices made by the compiler.

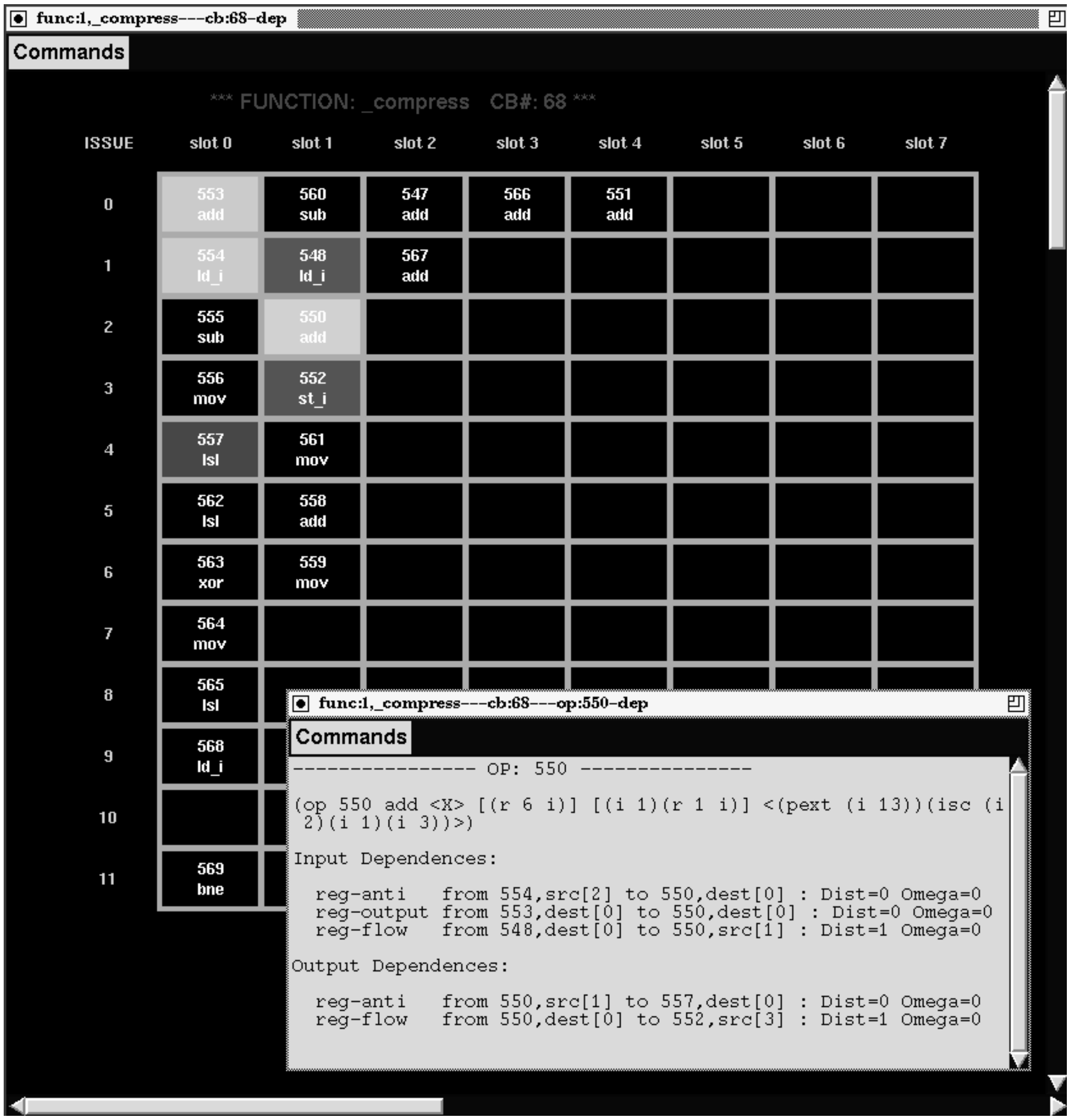
Creating graphical control block schedules is a simple process. First of all, a current function and control block within that function must be selected. Also note that the code in the current function must have been scheduled for a particular machine during compilation. Next, the user should select “Graphical Dependence Tool” from the control block pull-down menu. This will create a window in which the control block's schedule will be displayed. Spaces in the grid representing the control block will be filled with scheduled operations whose opcode strings and i.d. numbers will be displayed. The window and the graphics displayed may be resized in the same way the control flow and dependence arc graphs could be adjusted.



```
-----Dependence Distance Between Op:553 and Op:558-----


DEPENDENCE DISTANCE : 5

DEPENDENCE PATH :

  reg-flow   from 553,dest[0] to 554,src[2] : Dist=1 Omega=0
  reg-flow   from 554,dest[0] to 555,src[1] : Dist=1 Omega=0
  reg-flow   from 555,dest[0] to 556,src[0] : Dist=1 Omega=0
  reg-flow   from 556,dest[0] to 557,src[2] : Dist=1 Omega=0
  reg-flow   from 557,dest[0] to 558,src[0] : Dist=1 Omega=0

..for dependence arc types..

  reg flow
  reg anti
  reg output
```

Figure 3.6: Dependence arc graph.

Once the control block's schedule is produced on the screen, there are several interactive analysis tools that can be useful. The first feature is activated through use of the left mouse button. By single clicking on a grid square representing a particular operation, that operation and all operations with dependence relations will be highlighted. The original operation selected will be colored cyan. Operations dependent on the original will be colored in red, and operations on which the original is dependent will be colored in purple. Additionally, there are two distinct shades of red and two distinct shades of purple. Operations dependent on the selected operation which are forced into their scheduled execution cycle by the dependence arc length from the selected operation are colored a dark red. Dependent operations which are not forced into their scheduled execution cycle by the selected operation are colored a lighter red. Operations on which the original is dependent, whose connecting dependence arcs are of a cycle distance which forces the original operation into its scheduled execution cycle are colored a dark purple. Operations on which the original is dependent, but whose dependence arcs do not force the original operation into its scheduled cycle are colored a light purple. The information provided by this functionality provides a very efficient method of examining dependence constraints between scheduled operations. Using the left mouse button to double click on an operation provides the same graphical information; and, in addition, a window containing the operations Lcode text and descriptions of all incoming and outgoing dependence arcs will be created. Figure 3.7 shows an example of this tool's operation.


func:1,_compress---cb:68-dep
Commands
func:1,_compress---cb:68---op:550-dep
Commands

*** FUNCTION: _compress CB#: 68 ***

ISSUE	slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7
0	553 add	560 sub	547 add	566 add	551 add			
1	554 ld_i	548 ld_i	567 add					
2	555 sub	550 add						
3	556 mov	552 st_i						
4	557 lsl	561 mov						
5	562 lsl	558 add						
6	563 xor	559 mov						
7	564 mov							
8	565 lsl							
9	568 ld_i							
10								
11	569 bne							

```
---------------- OP: 550 ---------------

(op 550 add <X> [(r 6 i)] [(i 1)(r 1 i)] <(pext (i 13))(isc (i
 2)(i 1)(i 3))>)

Input Dependences:

  reg-anti   from 554,src[2] to 550,dest[0] : Dist=0 Omega=0
  reg-output from 553,dest[0] to 550,dest[0] : Dist=0 Omega=0
  reg-flow   from 548,dest[0] to 550,src[1] : Dist=1 Omega=0

Output Dependences:

  reg-anti   from 550,src[1] to 557,dest[0] : Dist=0 Omega=0
  reg-flow   from 550,dest[0] to 552,src[3] : Dist=1 Omega=0
```

Figure 3.7: Scheduled dependence constraint tool.

The middle mouse button provides another type of information. The middle mouse button can be clicked within a row representing an execution cycle, and operations issued in or before the selected cycle whose latencies push the issue time of dependent operations to a cycle after the selected cycle will be highlighted in blue. If the middle mouse button is double clicked in an issue cycle, the same operations will be colored blue, and the dependent operations in the following cycle will be colored grey. The user can use this feature to quickly identify the reasons for empty issue slots. A parameter can be set which allows this function to include zero cycle dependences in its analysis. In this case, operations with completion times greater than the selected cycle, which have dependent operations that can actually be scheduled in the same cycle, will also be identified by this procedure. The parameter used to set this option is named, “examine_zero_cycle_dependences”. It may be set in the STD_PARMS file or set interactively using the “Change Parameters” feature. Pressing the right mouse button over any part of the control block schedule will unhighlight all colored operations. This set of Lanalysis functionality can be very useful for researchers examining or implementing IMPACT scheduling heuristics.

3.4 Lanalysis Usage

There are three methods in which Lanalysis can be used. The first and usual method involves running the tool under the control of the graphical user interface. This usage is invoked by typing “lanalysis” at the UNIX prompt. All functionality of the tool is available when it is used in this fashion. The second method of using Lanalysis is to run the program without the control of the graphical user interface. This is accomplished by entering “Lanalysis” at the UNIX prompt. By running Lanalysis in this manner, the user forfeits all functionality which provide graphical output. However, all the commands which produce text output can be used at the Lanalysis command line. This method can be useful for users who do not need the extra functionality or cannot display graphics, and it can be useful while debugging new Lanalysis features. The last and more complicated procedure for using Lanalysis is to have it execute in a non-interactive fashion. This is accomplished by first setting the standard parameter *non_interactive_mode* to *on* in the STD_PARMS file. Next, a file must be created which contains a list of valid text-output Lanalysis commands. There should be one command on each line of the file. Finally, the user should execute Lanalysis such that the command file is fed to the program as input. This can be done using the "<" UNIX operator. Each Lanalysis command in the file will be executed, and the appropriate output will be produced. Non-interactive use of Lanalysis can also be achieved through the use of a script called "get_stats". This script will automatically run Lanalysis in the non_interactive mode. The script will allow the user to provide an Lcode file name or multiple file name specification, a STD_PARMS file name, and a command file name as arguments. This can be useful when examining multiple sets of Lcode files. When Lanalysis is executed using this script, the command file provided must begin with the line: "load SED_FILE_SPEC". This line will be automatically altered by the "get_stats" script in order to load all the files specified by the user.

4. IMPLEMENTATION OF LANALYSIS

The Lanalysis performance analysis tool is comprised of several separate program modules. The software designed for this tool include Lanalysis, lanalysis, and Ldag, which were briefly described in Chapter 3. This chapter will describe the implementation of each of these modules as well as the operation of the system as a whole. Lpretty is an IMPACT tool used in conjunction with Lanalysis; however it was developed separately. Therefore its implementation will not be described. Lanalysis and Ldag are written in the "C" programming language, and lanalysis is a script written in the Tk/Tcl programming language. The first section of this chapter will provide an overview of the Tk/Tcl toolkit. The following three sections will discuss the organization and implementation of the Lanalysis, lanalysis, and Ldag program modules. The final section of this chapter will provide an understanding of the interface and communication between the various Lanalysis modules.

4.1 Overview of Tcl and the Tk Toolkit

Tcl and the Tk toolkit are the main components of a software package developed at the University of California at Berkeley [8]. This package provides a programming system for the design and development of graphical user interface applications. The

Tk/Tcl tools were selected for the Lanalysis project because the use of these tools can greatly speed the development time for X Windows-based graphical user interfaces.

Tcl is a simple scripting language designed for controlling and extending applications. The name stands for “Tool command language.” Tcl scripts can include many of the usual programming constructs such as variables, arrays, if-then constructs, looping, and procedures. Programs written in the Tcl language can be interpreted and executed by a program called *wish* which is included in the Tk/Tcl package. The Tcl language allows these programs to run child processes and communicate with other programs. Tk is an extension to Tcl which provides an interface to the X Windows system. Commands in the Tk toolkit may be included in Tcl scripts and enable programmers to quickly implement Motif-like user interfaces. There are Tk commands which produce graphical buttons, pull-down menus, dialogue boxes, scrollbars, text, graphics, etc. The software, *lanalysis*, is a script written in the Tk/Tcl language. It is responsible for the graphical user interface to the Lanalysis tool and will be described in Section 4.3. For more information on the Tk/Tcl tools, see [8].

4.2 Organization of the Lanalysis Program Module

Lanalysis is the “C” program which can be considered the main module of the Lanalysis package. It is responsible for all the major analysis and calculation functions provided by the tool. This section will describe all the major features of the implementation of this program.

The Lanalysis program is currently divided into nine program files and one header file. These files include l_analysis.h, l_analysis.c, cbs.c, controlgraph.c, current.c, depschedgraph.c, load.c, loops.c, misc.c, and regions.c. The first file, l_analysis.h, contains definitions of the data structures used in the program as well as assignments of command codes for all the functions available in the Lanalysis tool. The file l_analysis.c contains the main loop of the program. The files regions.c, loops.c, and cbs.c contain functions which perform calculations of Lcode region, loop, and control block data, respectively. The file controlgraph.c holds functions which produce control flow graphs for the various Lcode constructs. The depschedgraph.c file contains functions that implement the scheduled dependence examination tools as well as the dependence arc graphing functions. Functions which read in and organize Lcode files are located in load.c. The file current.c provides functions for selecting and displaying Lcode constructs marked as "current" within the Lanalysis tool. Finally, the misc.c file contains extra functions used by Lanalysis which don't fall into a particular category, but are essential to the program's operation.

Lanalysis is a standard Lcode program. Like normal IMPACT Lcode modules, it is compiled with the IMPACT Lcode program files. Therefore, all the data structures defined within the IMPACT Lcode framework are available to the Lanalysis program; these standard data structures are used wherever possible. The entry point to the Lanalysis program is the function "L_gen_code", which is called from the IMPACT l_main.c file. L_gen_code can be considered the "main" function of the Lanalysis software, and this function contains the main program loop of the tool.

The main loop of the Lanalysis tool is a while statement which prompts the user for commands, interprets the commands, and calls functions which produce the requested output. Commands are read in from the standard input as strings, and are converted to a unique integer number by the "convertcom" function. A switch statement calls the desired functions based on the value of the command's integer code. In most cases, a function written specifically for each command is invoked from that command's area of the switch statement; however, in some cases a separate function is not used, and the command's actions are executed inside the switch statement. Once the desired function or functions produce their output, a special end marking string is emitted, the user is given another prompt, and the program waits for the next command. Special strings, such as the one just mentioned, are used to communicate with the graphical user interface and will be discussed in more detail in Section 4.5.

A convention concerning the naming of commands and functions was followed during the development of the Lanalysis program. Lanalysis user command names were chosen to reflect the functionality of the command. Some of these command names are quite long, but users are not required to memorize or even type these commands because they can be selected automatically through the use of the graphical user interface. This will be described in a later section. The user commands are named entirely with lower case characters. The corresponding integer command code labels were defined as the same word, but completely capitalized. Finally, the actual "C" language function names were chosen to be exactly the same as the corresponding lower-case user command names. It was not necessary to follow this naming convention, but with the great number of Lanalysis commands implemented, this convention does eliminate the potential for unnecessary confusion.

Much of the main functionality of the Lanalysis tool involves the collection and organization of Lcode based upon annotated profiling statistics. There are Lanalysis commands which create ordered lists of various Lcode constructs and display all or part of these lists to the user. These lists typically contain more information than is contained in the standard Lcode data structures. Therefore, additional Lanalysis data structures were defined for the implementation of these ordered lists.

There are currently four major categories of lists that the Lanalysis tool creates. These include lists of Lcode functions, regions, loops, and control blocks. Four data structures are defined to be used as nodes in linked lists which hold the collected information. The first structure created is used to implement lists of most frequently executed control blocks. In addition to a pointer to the actual Lcode control block data structure, this new data structure can hold the name of the function in which the block is located, the dynamic operation count of operations it contains, a cycle count calculated from scheduler information, and a sum of dynamic register spill operations. The data structure definition can be seen in Figure 4.1.

```
typedef struct Frequent {
    double              cbweight;    /* weight of the cb - sort by this */
    char               *funcname;   /* function block is in */
    L_Cb               *cb;         /* pointer to the control block */
    struct Frequent    *nextfr;     /* pointer to next most frequent */
    double              cyclecount;  /* cpu aprox cycle count */
    double              spillcount;  /* dynamic spill oper count */
} Frequent;
```

Figure 4.1: “Frequent” data structure.

Another data structure defined for use with Lanalysis lists holds information concerning Lcode program loops. This structure is designed to contain a pointer to the actual Lcode loop structure, the dynamic operation count inside the loop, the CPU cycle count calculated for the loop, the name of the function the loop is in, attributes associated with the function containing the loop, the name of the file the loop came from, a pointer to the loop header control block, the loop’s nesting level, the loop i.d. number, and the number of dynamic spill operations within the loop. This data structure definition can be seen in Figure 4.2.

```
typedef struct Looplist {
    double            dynopcnt;       /* dynamic op count (sort by this) */
    double            loopercent;     /* loop percentage */
    double            loopcycpercent; /* loop percentage by cycle count */
    char              *funcname;      /* name of func that loop is in */
    char              *filename;      /* file that function is from */
    L_Cb              *loopheadcb;    /* loop header cb */
    int               nesting_level;  /* loop nesting level */
    L_Attr            *attr;          /* function attributes */
    struct Looplist   *nextloop;      /* pointer to next loop */
    struct Looplist   *nextloopbycyc; /* next loop in CPU cycle count order*/
    int               id;             /* loop id number */
    L_Loop            *loop;          /* pointer to loop (added 10/18/94) */
    double            cyclecount;     /* cpu aprox cycle count */
    double            spillcount;     /* dynamic spill count */
} Looplist;
```

Figure 4.2: “Looplist” data structure.

The next data structure used by Lanalysis is designed to hold information for a list of Lcode regions. This structure contains the dynamic operation count with the region, the name of the function containing the region, and the region number. This structure’s definition can be seen in Figure 4.3.

```
typedef struct Regionlist {
    double              regionweight;  /* Dynamic weight of region */
    char                *funcname;     /* function region is in */
    int                 region;        /* region number */
    struct Regionlist   *nextregion;   /* pointer to next region */
} Regionlist;
```

Figure 4.3: “Regionlist” data structure.

The last major data structure is used to hold the list of Lcode functions which are loaded into the tool. This data structure is defined to include a pointer to the actual Lcode function structure, the dynamic operation count of the function, the i.d. number of the function in the ordered list of functions, the name of the file the function came from, the execution cycle count of the function, the number of dynamic spill code operations in the function, and pointers to lists of regions, loops, and control blocks contained in the previously defined structures. The definition for the "Functionlist" data structure can be seen in Figure 4.4.

```
typedef struct Functionlist {
    double              funcweight;     /* Dynamic weight of function */
    int                 funcnumber;     /* Number of function in list */
    L_Func              *func;          /* pointer to function structure */
    Regionlist          *regionlist;    /* pointer to list of regions */
    Looplist            *loops;         /* pointer to list of loops */
    Frequent            *freqlist;      /* pointer to list of most freq cbs */
    Frequent            *docfreqlist;   /* list of most freq cbs by dynopcnt*/
    Frequent            *cycfreqlist;   /* list of cbs by approx cycle count*/
    char                *filename;      /* file function came from */
    struct Functionlist  *nextfn;        /* pointer to next function */
    struct Functionlist  *nextfnbycyc;   /* used for cycle count order list */
    double              cyclecount;     /* cpu aprox cycle count */
    double              spillcount;     /* dynamic spill oper count */
} Functionlist;
```

Figure 4.4: "Functionlist" data structure.

The function, region, loop, and control block data structure types are organized in memory in a way which facilitates logical access to the stored information. The entry point to the entire data structure is a pointer to the linked list of function information nodes. This list begins with a "dummy" node and is followed by nodes created for each Lcode function read into the Lanalysis tool. Each one of these function nodes contains pointers to linked lists of the region, loop, and control block structure types which hold information related to the regions, loops, and control blocks contained inside the particular function the node represents. The linked lists for regions, loops, and control blocks are also headed with a "dummy" node. Information held in the "dummy" nodes usually reflects a sum of the statistics held in the remainder of the list. Inspection of the

Lanalysis source code will provide detailed information on how information is collected and stored in this data structure. In addition to the data structure just described, three additional linked lists are created by the Lanalysis tool. These are ordered lists of regions, loops, and control blocks, but members of these lists are taken from all the functions loaded into memory rather than one particular function. The information in these lists is used when collecting data for entire programs instead of single functions. A block diagram of the major Lanalysis data structures can be seen in Figure 4.5.

There are several special functions written for the Lanalysis tool. These include functions that replace existing IMPACT or “C” functions, as well as some important additional functions. A function called “print2” replaces the usual “C” language “printf” statements in cases where the text being printed is the output of an Lanalysis user command. The “print2” function behaves the same as “printf”, but if there is an output file started by the user, this function will print to that file as well as the standard output. The standard IMPACT “L_warn” and “L_punt” functions have been replaced with functions which additionally produce control strings which communicate with the graphical user interface. The same has been done for the “M_punt” function. Finally, there is a function named “reset_tcl” which places the Tk/Tcl graphical user interface into its original starting state. This function is used when the Lanalysis tool is restarted by the user.

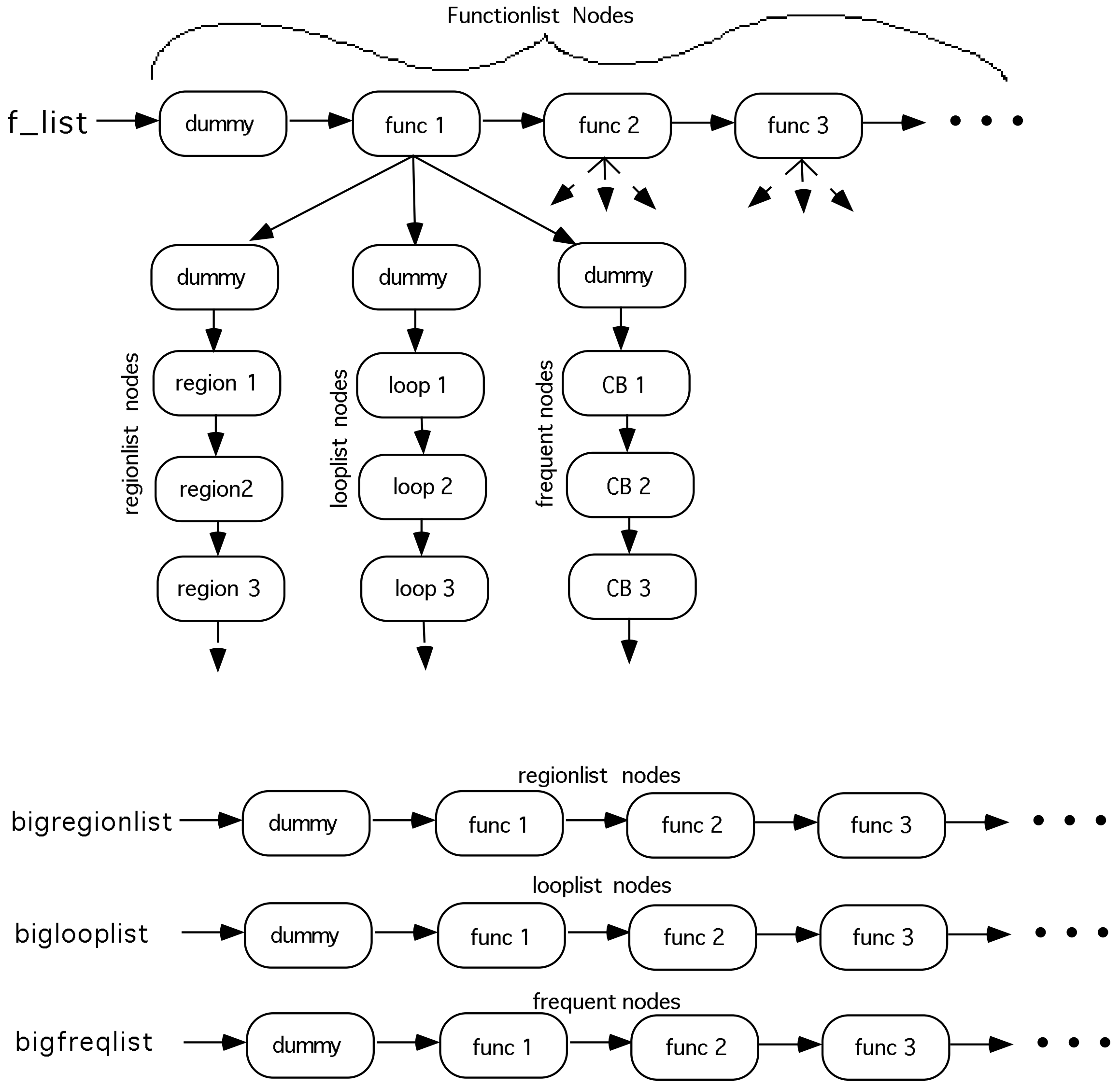


Figure 4.5: Lanalysis data structure organization.

4.3 Organization of the “lanalysis” Tk/Tcl Script

The Tk/Tcl script *lanalysis* is responsible for implementing the graphical user interface to the *Lanalysis* program module. The Lanalysis graphical environment includes many pull-down command menus, status indicators, and a scrolling text window for Lanalysis output. Additionally, the *lanalysis* script creates and displays all of the tool's non-textual interactive functionality. This section will describe the organization of this Tk/Tcl script.

The *lanalysis* script is interpreted by a program called *wish* which is included in the Tk/Tcl software package. The first line of the executable script will invoke the *wish* program. This first line should be configured with the correct location of the *wish* executable for the particular machine and environment the user is working with. The remainder of the script will read by *wish* as input and will define the graphical user interface.

The first portion of the *lanalysis* script creates the Lanalysis display. To begin with, all the fonts and colors used by the program are specifically defined so that they remain constant across various workstation platforms. From this point on, any reference to a particular color or font should be made through the defined Tcl variable. Once this is done, the layout of the Lanalysis main screen is created. The "current" status labels, pull-down command menus, main text window and scroll bars, etc. are all specified and placed using simple Tk/Tcl commands.

After the Lanalysis display has been built, many Tcl procedures are defined which are responsible for the functionality of the interface. The important procedures will be described shortly; but first, the very end of the *lanalysis* script will be discussed. The last couple lines of the Tk/Tcl script invokes the main *Lanalysis* executable and creates a UNIX pipe to and from the program's input and output. This will serve as the main communication link between the two program modules. The graphical user interface will send Lanalysis commands to the main program and will receive and display the output it generates. Details of the communication between Lanalysis modules will be described more fully in Section 4.5.

The most important *lanalysis* procedures include those which send and receive information to and from the *Lanalysis* program, the procedures which create control flow and dependence arc graphs, and the procedures which implement the scheduled dependence constraints examination tool. The procedures “invoke”, “menex”, “menex2”, “invoke1”, “invoke2”, and “menexargs” are all designed to send commands to the main *Lanalysis* program and then process the received output. The “invoke” procedure grabs the text at the user prompt and sends it to the *Lanalysis* pipe. It then places the generated output in the main text window. This procedure also scans the output for certain control strings which will be processed but not displayed in the text window. Section 4.5 will include a full description of such control string sequences. The procedures “menex” and “menex2” are usually called when a user selects a command from the pull-down menus. These procedures actually insert the appropriate command strings at the Lanalysis prompt, and then call the “invoke” procedure to have them executed. The “menexargs” function creates a window which prompts the user for an additional argument to a command, and then calls the “menex2” procedure which sends the original command and its argument into the execution pipe. The “invoke1” and “invoke2” procedures are equivalent to the “menex” and “menex2” functions, but the commands themselves are not printed to the Lanalysis screen. These functions are useful for sending Lanalysis commands that produce output used by the interactive graphical tools. These commands are typically sent repeatedly as the user interacts with various tools; therefore, displaying these commands or their output would merely fill the main text window with useless information.

The next major procedure is called “graphics”. It is responsible for creating and manipulating directed graphs. These graphs are of two types: the control flow graphs and the dependence arc graphs. The “graphics” function sends a command to the *Lanalysis* module requesting the specification for a particular graph. This specification is then sent to the *Ldag* module for processing. The *Ldag* program calculates coordinates for the layout of the graph which are fed back to the *lanalysis* "graphics" procedure which creates the display. Again, details of the communication between Lanalysis modules will be covered in Section 4.5.

The "graphics" procedure takes three arguments: the command sequence that will be sent to the main Lanalysis program, the name of the window that will hold the graph, and a flag identifying the graph as a control flow graph or as a dependence arc graph. There are several Tk/Tcl functions which are used to construct the first argument to the "graphics" procedure when the Lanalysis command requires extra arguments or configurations. These extra functions are required particularly when creating control flow graphs to a specified nesting level.

The "graphics" procedure begins by sending its first argument to the *Lanalysis* program. Next, the window which will contain the graph is constructed. A pull-down menu containing commands for resizing the graph image and for producing Postscript output of the image is created, as well as the scrollable canvas on which the graph will be plotted. Once the display window is created, the procedure feeds the output generated by *Lanalysis* to the *Ldag* program and then reads in the generated graph layout. The graph layout will be a list of nodes and arcs with sets of coordinates specifying their placement on the graph canvas. Predefined strings and characters will identify the type of node or arc to be displayed, and the "graphics" procedure will color code or format the output accordingly. These type identifier strings will be described in Section 4.4, which covers the *Ldag* program module.

Inside the loop which processes the output of the Ldag module, the bindings for mouse click actions are defined for each node in the graph. The mouse functionality will depend on the type of graph being produced. For control flow graphs, the left and middle mouse buttons will be active and execute commands depending on the type of node selected. Pressing the left button on a node representing a control block will send commands to *Lanalysis* and execute functions to produce a resizable text window which will contain the control block's Lcode returned from the main program. This Lcode will be formatted using the Lpretty print template which is currently active. Pressing the left mouse button over a node which represents a collapsed loop of a greater nesting level will send Lanalysis commands and call procedures which produce another window containing the control flow graph of that loop to the next nesting level. Selecting such a node with the middle mouse button will produce a similar result, but a procedure which prompts the user for the number of nesting levels to display will be invoked instead of defaulting to one additional level. If a dependence arc graph is being generated, all three mouse buttons will have actions bound to them. All nodes in this type of graph represent individual operations within a control block. Selecting a node with the left mouse button sends commands to Lanalysis and calls a procedure to display the operation's dependence information in a text window. The middle mouse button will send Lanalysis commands and call the procedure "depheight" which displays the dependence chain between two operations. The right mouse button executes commands that clear all nodes highlighted by the middle mouse button use. See Chapter 3 for details of the tools functionality from the user's point of view.

The final major *lanalysis* procedure is called "util_graph". It is responsible for implementing the scheduled dependence constraint analysis tool. This tool creates a graphical representation of the current control block's schedule and allows dependence arcs between scheduled operations to be interactively examined. The function "util_graph" takes two arguments which are the Lanalysis command name to be executed and the name of the window in which the control block's schedule will be displayed. The procedure begins by sending its first argument as a command to the *Lanalysis* module. Next, the window containing the tool is defined and displayed. A pull-down command menu similar to that used in the “graphics” procedure is created for resizing the image and creating Postscript output. The canvas and scroll bars are specified similarly as well. Once the window is in place, the output of the *Lanalysis* program is processed creating the graphical schedule. First, the height and width of the control block’s schedule are read by the procedure, and text labels for the issue slot columns and execution cycle rows are created. Next, a doubly nested for loop fills the issue slots for all execution cycle rows with the scheduled operations. Each line received from the main program’s output contains the Lcode operation’s i.d. and opcode which will be placed in the graphical schedule. If a slot is not filled by the IMPACT scheduler, *Lanalysis* will output “empty” on the line which corresponds to that particular issue slot.

While the loop executes, filling execution slots with scheduled operations, bindings for mouse actions are defined for each issue slot. Functionality is defined for all three mouse buttons. Pressing the left mouse button over a scheduled operation will send commands and call procedures which highlight operations connected by dependence arcs. Double clicking the left mouse button on an operation will additionally execute commands to create a text window which will contain descriptions of these dependences. Activating the middle mouse button over a particular issue time row will call procedures to highlight operations whose dependence arcs cross that issue time. Double clicking the middle button will additionally cause the operations at the ends of these arcs to be identified. The right mouse button will clear all highlighted operations. Chapter 3 describes the usefulness of these features in more detail. These mouse button bindings typically involve sending *Lanalysis* commands which will return operation i.d. numbers to highlight. Then Tk/Tcl procedures which change the background color of rectangles in the schedule grid are invoked to highlight the specified operations.

4.4 Organization of the Ldag Program Module

Ldag is the Lanalysis module which calculates the layout for both control flow graphs and dependence arc graphs. This program is actually a modified version of "xmdag" written by TUBITAK Software. This free software was obtained from the internet. It was necessary to modify the "xmdag" program for use with the Lanalysis tool, and the resulting program was named *Ldag*. This section will describe the changes and enhancements made to the original software and will describe the operation of the *Ldag* program.

Xmdag was originally designed to create directed acyclic graphs from a specification of nodes and their children provided as input to the program. The graph would be displayed using X Windows graphics commands. Changes in the *Ldag* program include the replacement of the original program's inefficient graph levelizing algorithm, the addition of the ability to handle directed *cyclic* graphs, the addition of the ability to differentiate between different "types" of directed arcs, and the replacement of the program's graphical output with a text-based output that can be read by the *lanalysis* module.

The original graph levelization algorithm provided in the "xmdag" program was grossly inefficient. Levelization of very complicated IMPACT control flow structures could require hundreds of megabytes of memory and as much as an hour to complete. The reasoning for the original choice of algorithm was never fully understood, but it was apparent that a different algorithm would be needed for use with the Lanalysis tool. The replacement algorithm was a recursive depth first search function which could levelize even complicated graphs extremely quickly.

The capability to handle directed graphs which contain cycles was achieved in two different ways. The first was to have *Lanalysis* label backward branching arcs as such and actually have *Ldag* treat them as forward arcs during the layout of the graph. In this case, *Lanalysis* would list the parent node as a child of the real child node, and mark it with a character signaling the reverse relation. The arc would be labeled as a "back edge" by the *Ldag* program, and when it is displayed by the *lanalysis* script, the arrows indicating the arcs direction would be reversed and the color of the arc would be changed. This was acceptable for most cases of cyclic control flow graphs, but there were cases in which *Lanalysis* could not detect and label backward arcs which caused the *Ldag* program to fail. Therefore, the *Ldag* module was enhanced so that it could detect and treat backward branching arcs on its own. During levelization of the graph, if an arc creating a cycle is detected, the parent node of the arc will be added to the children list of the child node, and the child node will be removed from the children list of the real parent node. The new arc in the children list of the child node will be marked as a "back edge," but will be treated as a forward branch by the layout algorithm. With this change, *Ldag* can handle graphs containing cycles whether they are marked or not. The original method for graphing backward branching arcs was left in place because backward branches labeled by *Lanalysis* are truly backward loop branches. *Ldag* would produce correct output of graphs without the "back edge" markings from *Lanalysis*, but the layout of the graphs could differ from what would be naturally visualized since the selection of edges to be marked as backward branching would depend on the order in which nodes are visited by the depth first search.

In addition to marking arcs as backward and forward edges of a graph, the need to label multiple types of edges was encountered as a result of creating dependence arc graphs. Currently, Lanalysis can display eight types of dependences in a directed graph. The *Ldag* program has to hold "type" information for each edge in the dependence graph. For dependence arc graphs, *Lanalysis* will identify the dependence type for each edge in the graph. *Ldag* will carry this information, and the list of arcs output to the *lanalysis* script will contain labels identifying graph edge types. The actual format of the labels used in the input and output to *Ldag* will be described later in this section.

The final change made in the *Ldag* program to be mentioned is the replacement of X Windows graphics output with the output of text that will be processed by the “graphics” procedure in the *lanalysis* script. *Ldag* will produce a list of graph nodes with coordinates and a list of graph arcs with types and coordinates as its output. The format of the program’s output will be discussed shortly.

The *Ldag* program module accepts a graph specification in a simple definition format, calculates the layout of the graph, and encodes this layout in a simple output format. The input to the *Ldag* program is a file which contains a list of nodes and their children. Nodes may be named integers or strings, with the following exceptions. The node names may not begin with capital or lowercase letters “A” through “I”, and they may not begin with an asterisk. These characters are used by *Ldag* as markings of graph edge types. The Lanalysis tool will only label nodes with integers, with the exception that nodes representing collapsed loops in a control flow graph will consist of an “L” followed by an integer i.d. Each line of the input file will contain a node label followed by a colon which is followed by a list of children nodes separated with commas. The line will be terminated with a semicolon. Spaces should not be included in any part of any input line. The special characters, upper and lowercase “A” through “I” and the asterisk, may be placed immediately in front of a node label in the children list of a parent node. When *Ldag* is used to create control flow graphs, the asterisk is the only necessary special character. Placing an asterisk in front of a node in the children list of a parent indicates that the child is really the parent in the graph and that the edge between the two nodes is a backward branching arc. The edges between parent nodes and unmarked children nodes will be treated as forward branching arcs. When *Ldag* is used to create dependence arc graphs, the upper and lowercase letters “A” through “I” are used to label arcs to all children nodes. The letters “A” through “I” are used to identify arc types in a dependence arc graph. Capital letters are used to label forward going edges, and lowercase letters are used to label backward going arcs in the same way that the asterisk was used for control flow graphs.

The output generated by the *Ldag* program is formatted to be processed by the “graphics” procedure in the *lanalysis* script. Each line of output generated for a directed graph consists of a type identifier word followed by four numbers separated by spaces. Each output line represents either a node or an arc of the graph, and the set of four numbers is interpreted accordingly. If the line represents a graph node, the identifier word will be “NODE”. The following two numbers are the coordinates of the upper left corner of the box which will represent the node in the graph. The next number is the i.d. label for the node. The last number is usually zero unless the node has an outgoing arc to itself. In this case, the fourth number will be an integer between one and ten which signals the type of the self-connecting arc. If the *Ldag* output line represents an arc, the identifier word will be “FORWARD” or “BACK” for control flow graphs. These will define the arc as a forward or backward edge. For dependence arc graphs, the identifier word could be “ARC_A” through “ARC_I” or “BARC_A” through “BARC_I”. These identifiers will define forward and backward arcs of types “A” through “I”. For all lines representing arcs, the first two numbers are the coordinates of the beginning of the arc, and the last two numbers are the coordinates of the end of the arc. After all node and arc definitions are output by the program, *Ldag* produces a final line containing the string “*ENDOUT” which signals the end of the output. An example *Ldag* input and output are shown in Figure 4.6. Note that type identifiers for both control flow and dependence arc graphs are used in this example in order to demonstrate their operation. The input file used here would not be created by *Lanalysis* for any actual case.

```
1:2,3;
2:3,b4,*5;
3:*3,4,5;
4:D5;
```

(a)

```
NODE 400 15 1 0
NODE 400 115 2 0
NODE 500 215 3 10
NODE 300 315 4 0
NODE 400 415 5 0
FORWARD 414 45 414 115
FORWARD 414 45 499 115
FORWARD 414 145 514 215
BARC_B 411 145 296 215
BACK 417 145 402 215
FORWARD 499 115 514 215
BARC_B 296 215 311 315
BACK 402 215 402 315
FORWARD 514 245 314 315
FORWARD 514 245 499 315
ARC_D 313 345 413 415
BACK 402 315 417 415
FORWARD 499 315 414 415
*ENDOUT
```

(b)

Figure 4.6: Ldag communication files: (a) input, (b) output.

4.5 The Interface and Communication Between Lanalysis Modules

There are four program modules which work together to implement the Lanalysis performance analysis tool. These are the programs *Lanalysis, lanalysis,* and *Ldag*, along with the additional IMPACT program *Lpretty*. These four modules communicate with each other in a fashion that is automatic and transparent to the Lanalysis user. This section will describe the communication links between each module of the tool.

Most of the communication between Lanalysis modules takes place between the *Lanalysis* program and the *lanalysis* script. These two modules send information back and forth frequently during normal use of the tool using the graphical user interface. As described in Section 4.3, the *lanalysis* script opens a UNIX pipe to and from the *Lanalysis* program's input and output. This pipe is treated as a file which is assigned to a

Tcl variable. The *lanalysis* script writes to and reads from this file in order to communicate with the main program. The typical communication sequence is as follows. The script writes an Lanalysis command to the UNIX pipe file. This is received as the standard input by the *Lanalysis* program. *Lanalysis* produces its output and writes it to the standard output, which in this case is the communication pipe. The *lanalysis* script reads the output and prints it to the user screen until the special "*ENDOUT" string is reached. This string is placed at the end of the output by the *Lanalysis* module.

While the output is read and placed on the text screen, it is also being scanned for special purpose strings. If the string "*@CONTROL@*" is encountered, the next line of output will be read, but it will be evaluated as a Tcl command script. Through the use of this string, the *Lanalysis* program can send commands which can perform any action within the graphical user interface. Multiple Tcl commands may be sent separated by semicolons, but they must all reside on one line of the output. In several cases, *Lanalysis* sends text which is not meant to be displayed in the main text window of the tool. In these cases, temporary UNIX files are created to hold the information, and the *lanalysis* script will open these files to access the data. In such cases, the *Lanalysis* program has to communicate the name of the temporary file created so that the script can access it correctly. When the script encounters the string "*@FILENAME@*", the next line of output will be read into the Tcl variable "tmp_file_name". The next line provided by *Lanalysis* would be, of course, the name of the file created. The *lanalysis* script can then use the name stored in the variable to open the temporary file. The "*@CONTROL@*" string may be used to achieve the same result as the "*@FILENAME@*" process and since *any* Tcl commands may be sent to the user interface, it is the only control string required to communicate with the Tcl program. The "*@FILENAME@*" control string is left as an artifact of an earlier version of the tool, but it is much less flexible than its successor.

The use of temporary files is especially prominent in the interactive graphical functions. In some cases, several temporary files are opened and written to at once, and the graphical user interface is sent several file names in order to open the files. After the information is read from the files and processed, the *lanalysis* script removes the files from the disk. Temporary file communication is used by *Lanalysis* to provide specifications for graph creation, lists of graph entities to highlight, text that will be displayed in windows other than the main text screen, titles for control flow and dependence arc graphs, as well for providing input files for *Ldag* and *Lpretty*.

Temporary files are created to hold the input for both the *Ldag* and *Lpretty* program modules. When *Lpretty* is used, *Lanalysis* provides the input in a temporary file. Then it executes *Lpretty* so that its output can be read out of a UNIX pipe and be processed by the tool. After the *Lpretty* program executes, the pipe is closed, and the temporary input file is deleted. The input to the *Ldag* module is also provided in a temporary file. *Lanalysis* creates and fills this temporary file and then communicates the file name to the *lanalysis* script. The script executes *Ldag* using the temporary file as input, and opens a pipe from its output. The graphing procedure reads the output from this pipe and displays the produced graph. The pipe is then closed and the temporary file is removed. Figure 4.7 displays the communication interactions between all four Lanalysis modules.

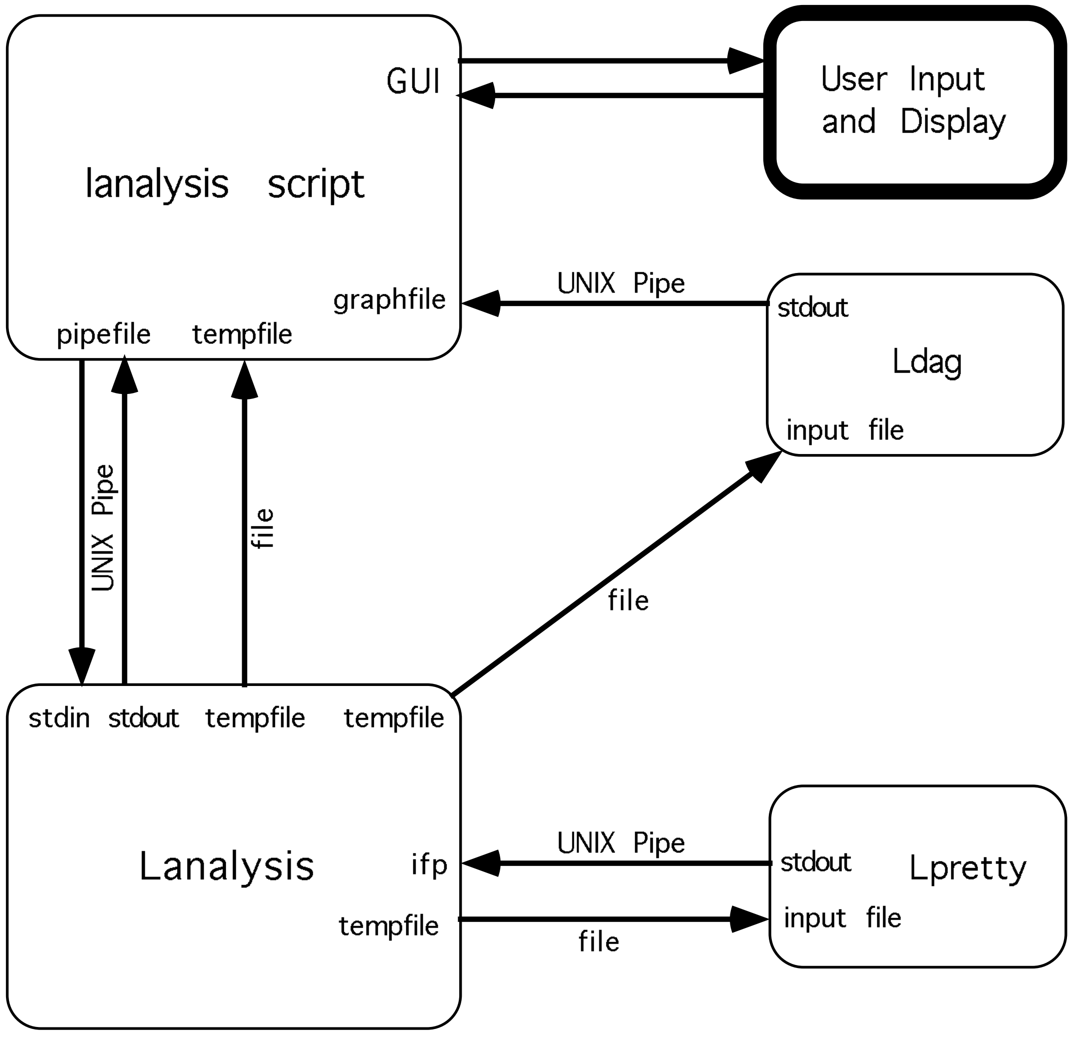


Figure 4.7: Program communication paths.

5. ADDING FEATURES TO LANALYSIS

The program modules designed for the Lanalysis performance analysis tool were designed to easily accommodate the addition of new functionality. As the Lanalysis tool becomes more popular with researchers working with the IMPACT compiler, the need for additional functionality will undoubtedly become apparent. This chapter will provide a procedure for creating new Lanalysis commands and will describe existing Lanalysis features which can be useful to future extensions to the tool.

5.1 Software Design for Expandability

There are several key design choices which promote the ability to quickly make enhancements to the Lanalysis software package. First, the design of the main *Lanalysis* program loop as a large “C” switch statement allows for the easy addition of extra command cases. The programmer merely has to add a case to the switch statement for each new Lanalysis command. Second, the use of the Tk/Tcl tools for creating the graphical user interface enables the programmer to rapidly define new pull-down menu command options and eases the creation of more complicated graphical tools. Finally, the design of the communication link between the graphical user interface and the main *Lanalysis* program allows the output of most tool functions to be implemented with simple printing statements which can be fed directly to the user interface display. Also, the ability for the *Lanalysis* program to send Tcl commands to directly define variables and control virtually any aspect of the user interface ensures that a communication barrier will never prevent exotic functionality from being created.

5.2 A Procedure for Adding Standard Lanalysis Commands

The addition of new commands to the Lanalysis tool is an extremely simple process. This section will describe a detailed procedure for augmenting the Lanalysis tool with new user commands.

The first thing that should be done when creating a new Lanalysis command is to choose a unique name for the command. The name should reflect the functionality of the command, and it is beneficial if the name is not too complicated if it is a command that users may want to manually enter at the Lanalysis prompt. Once the command name is selected, the numerical code for the command has to be defined in the "l_analysis.h" header file. This should be done in the section of the file labeled "command codes." The actual code number can be one greater than the code for the last command created, and the defined identifier should be the name of the command completely capitalized. This is done to remain consistent with the naming convention discussed in Section 4.2. Once the command code is defined, another "else if" clause has to be added to the "convertcom" function in the "l_analysis.c" program file. This clause will contain code that will return the command's numerical code if the command string is input to the tool. Next, an additional case should be added to the switch statement in the main loop of the program. The identifier for the command code should be used as the "value" of this new case. The functions that will be called for this new command will be placed in the newly created case area of the switch statement. In most cases, one "C" function named exactly like the name of the command is called to produce the desired output. Consequently, the next task is to create the actual function that implements the result of the new command. If the output of the new command is merely textual information, the text should be created using the “print2” function. As previously described, this behaves exactly like the standard “printf” function, but prints to an additional file if an output file is being written to by the tool. If the new command is designed to interact with the graphical user interface, control strings may be sent to the *lanalysis* script, temporary file communications may be needed, and the appropriate processing procedures may have to be created in the *lanalysis* script. See Section 4.5 for more details. The last step in the addition of new Lanalysis commands is to place the command in one of the user interface pull-down menus. This is easily accomplished by adding a new line to the appropriate menu definition. If the new command is part of a larger interactive tool, it probably does not require an entry in the pull-down menus. For example, the command which displays a control block when its node is clicked on in a control flow graph does not appear in any menu, and users are not expected to execute this command from the user prompt. Commands that can be entered manually by a user should also be described in the “printhelp” function located in the “misc.c” program file.

5.3 Useful Lanalysis Features

In addition to the expandability traits mentioned in Section 5.1, several features currently exist that may be useful to additional functionalities of the Lanalysis software. These features are all part of the *lanalysis* script module, and could be useful to future interactive analysis tools.

A Tcl procedure called “textwindow” is currently used by multiple tools and will certainly be useful to additional analysis functions. This procedure creates a resizable window in which any type of text can be printed. The procedure will open the temporary file indicated by the “tmp_file_name” Tcl variable and place the text contained in the file in the newly formed window. This procedure could be used whenever auxiliary information is requested that does not have to be displayed in the main Lanalysis screen.

Another group of reusable Tcl procedures are those which fill particularly tagged graphics entities with specified colors. These are used to highlight boxes in the dependence arc graphs and graphical control block schedules. When these graphics entities were created, they were tagged with unique i.d. numbers. *Lanalysis* can send a list of i.d. numbers through a temporary file, and these procedures will highlight the indicated objects with the specified color.

The last multipurpose Tcl procedure worth mentioning is the “graphics” procedure which creates the control flow graphs and dependence arc graphs. This procedure is reusable in the sense that it will create directed graphs that could represent anything. Any graph with nodes and directed edges between them can be displayed by the “graphics” procedure.

6. CONCLUSIONS AND FUTURE WORK

Lanalysis is an interactive performance analysis tool designed to analyze the output of the IMPACT C compiler. Through the use of this tool, researchers are able to efficiently examine important aspects of code produced by the compiler, and compiler writers are provided with a powerful tool for evaluating the performance of new or experimental compiler optimizations. As a result, both code analysis time and IMPACT development speed can be greatly enhanced.

This thesis introduced the Lanalysis tool. The complete functionality of the program was documented in the third chapter, which should be viewed as the Lanalysis user’s manual. The fourth and fifth chapters of the thesis provided a detailed description of the tool’s implementation and information concerning the simple expandability of the tool’s functionality. These two chapters are intended to be very useful to programmers who choose to create future extensions to the Lanalysis tool.

Many ideas for additional Lanalysis features have been conceived, but have yet to be implemented. This thesis will end with the presentation of several ideas for functionalities that would further augment the already useful performance analysis tool. Some of these ideas will require enhancements from the compiler, but others do not and can be implemented purely within the Lanalysis software.

The ability to display Lcode function call graphs was an anticipated addition to the Lanalysis tools. This capability would be extremely simple to implement because the existing procedure for displaying control flow graphs and dependence arc graphs could easily be used to create graphs representing function call structures. This, however, would require that the functional call information be annotated by the compiler in the Lcode read by Lanalysis. When this information becomes available, Lanalysis could quickly be adapted to present these graphs.

Another useful addition to the tool would be an improvement to the directed graph display procedure. In some cases, the layout of complicated control flow structures provided by *Ldag* can be quite congested which can cause their viewing to be difficult. One solution to this problem could be to add the ability to interactively rearrange the placement of nodes in the graph by hand. The flexibility of the Tk/Tcl programming language should enable this sort of ability to be implemented entirely in the *lanalysis* script module.

An extremely important enhancement to Lanalysis would be the added ability to retrieve and display the exact location inside the “C” language source file which led to the production of selected Lcode operations. Viewing these lines of “C” source code would enable users to better understand the purpose of the Lcode operations generated by the compiler. This functionality would require that source file line numbers be annotated to operations in the Lcode files.

In some cases, it would be nice to make comparisons between the code produced for different machine architectures. Presently, the Lanalysis tool can only examine code produced for one machine description during a single session, based on the architecture and machine description specified in the standard parameter file. The ability to specify these Lcode configurations from within the Lanalysis tool would enable users to make side by side comparisons of Lcode produced for different machines. This would be very useful for researchers evaluating new architectural features and would provide insights into the impact of machine architecture on compiled code.

A final thought for an Lanalysis enhancement would be the addition of the ability to automatically execute IMPACT compilation steps from within the tool's user interface. This would allow users to interactively perform optimizations on code loaded into the tool. The results of optimizations or compilation steps could then be immediately examined and evaluated.

The creator of Lanalysis hopes that the tool's growing use will continue in the future and hopes that the tool will constantly evolve to meet the ever-changing needs of IMPACT researchers in the years to come.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture,* Toronto, Canada, May 1991, pp. 266-275.

[2] G. E. Haab, "Design and implementation of a data dependence analyzer for fortran programs in the impact compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[3] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[4] W. W. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," *IEEE Transactions on Computers*, vol. 41, pp. 1537-51, December 1992.

[5] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringman, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229-248, January 1993.

[6] P. P. Chang and W. W. Hwu, "The Lcode language and its environment," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, Internal Report, April 1991.

[7] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[8] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Reading, MA: Addison-Wesley Publishing Co., 1994.