EMULATION OF THE INTERMEDIATE REPRESENTATION IN THE IMPACT COMPILER

BY

QUDUS BABATUNDE OLANIRAN

B.S., University of Illinois at Urbana-Champaign, 1997

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, 1998

Urbana, Illinois

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. His effective teaching style influenced my decision to pursue a graduate degree in computer architecture. I thank him for giving me the opportunity to pursue this goal. I wish to thank the entire IMPACT group for the infrastructure that created this research opportunity. For the great team environment on the X86 emulator project, I would like to thank John Sias and Michael Thiems. Teresa Johnson and John Gyllenhaal provided great leadership in the development of the project. Thanks to Mattew Merten for keeping the office lively. I am very grateful for all the help provided by Dan Connors. His influence on my education at the university has been invaluable. He has served as my mentor and provided many useful insights into the development of this project and thesis. For financial support through a fellowship, I would like to thank the University of Illinois Graduate College.

I would also like to express my appreciation to my parents for all their encouragement and support throughout my college years. Finally, I wish to thank and express my love to Heather Lee who has encouraged me throughout graduate school and for providing invaluable help with editing this thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

All compilers that support multiple target processors will typically use an intermediate representation to denote the converted or optimized code. One of the important aspects of this intermediate representation is that it provides the flexibility of generating code for a nonexistent architecture, i.e., research architectures with no hardware support. To support an architecture, a code generator is required to convert the intermediate representation to assembly language for a target processor. This invariably means that in order to test a research (or experimental) architecture, a code generator must exist for the development platform being used. This platform-specific requirement can be quite constraining for investigating new architectural features because some of the established conventions of an architecture only reside within the internal documents of the architecture's manufacturer.

One way to establish platform independence for an intermediate representation is to translate the low-level intermediate representation to a higher-level language structure.

This thesis presents a research infrastructure that provides high-level emulation capability for the internal representation of the Illinois Microarchitecture Project Utilizing Advanced Compiler Technology (IMPACT) compiler. In this research, the compiler's intermediate representation is converted to C programming language statements. C is a well-understood language that provides the desired architectural independence since it can be compiled on any platform.

Some of the main benefits of emulating intermediate representation in C include flexible profiling capabilities, cross-platform independence, and enhanced debugging capabilities. The most important of these benefits for the IMPACT compiler is profiling. Profiling has been shown to be successful in guiding code optimizations [1], [2], such as branch prediction strategies [3], [4], [5], and instruction-level parallelism enhancing optimizations. In order to get profiling information, probing instructions are inserted into the program code. Generally, an in-depth knowledge of the development architecture is required to correctly support efficient profiling facilities. The significance of the approach being presented in this thesis is that minimal knowledge of the architecture is required for generating profile information about a program.

This thesis illustrate some of the benefits of emulating the intermediate representation of a compiler in a high-level language. Its goal is to describe the emulation process, the surrounding environment, and the support for architectural features that are nonexistent in current processor technology.

Chapter 2 presents an overview of the IMPACT compiler used throughout this thesis. It also describes the structure of the low-level intermediate representation in the compiler. Chapter 3 describes the IMPACT simulation environment, the emulation environment developed for this thesis, and the complexity involved in both environments. Chapter 4 examines the benefits of emulating the IMPACT intermediate representation in a high-level language. Chapter 5 describes the implementation details crucial to the development of the emulation framework. Chapter 6 presents how the emulation tool is used. Finally, chapter 7 contains the conclusion.

2. BACKGROUND

One of the goals of the IMPACT compiler is to effortlessly provide portability in compilation for different development platforms. Emulating the intermediate representation in a high-level language can be instrumental in achieving this goal since high-level languages are portable by nature [6]. For this reason, an overview of the IMPACT research framework is discussed. The framework consists of two separate parts, the IMPACT compiler and the IMPACT simulator. The IMPACT simulator is discussed in chapter 3. In order to describe the emulation process, it is also important to discuss the general structure of the IMPACT low-level intermediate representation. Thus, in addition to the overview of IMPACT, this chapter presents a brief overview of the low-level intermediate representation. Detailed information about the intermediate representation and its key internal data structures can be found in [7].

2.1 Overview of the IMPACT Compiler

The IMPACT compiler has been recognized for its ability to effectively utilize profile information within speculative and predicated execution compilation techniques. A block diagram of the IMPACT compiler is presented in Figure 2.1. There are three levels of intermediate representation within IMPACT, *Pcode*, *Hcode*, and *Lcode*, which divides the compiler into distinct parts.

Pcode is the highest level of intermediate representation based on a parallel C representation with loop constructs intact, thus closest to the source code. Within *Pcode*, the IMPACT compiler performs loop transformations [8], function inlining [9], profiling, and dependence analysis [10], [11]. *Hcode* is simply a flattened C representation, which serves as the bridge between *Pcode* and *Lcode*. In fact, *Hcode* can be regarded as an extension of the *Pcode* format.

Lcode is the lowest level of program intermediate representation and is in the form of a generalized register transfer language. It can be viewed as an instruction set for a virtual load-store architecture. All machine independent optimizations [12] are applied at this level. Likewise, advanced compilation techniques, such as superblock [13] and hyperblock [14] formation, are performed on the *Lcode* representation of programs. The focus of this thesis is to emulate the *Lcode* representation in C and provide architectural independence in the profiling process.

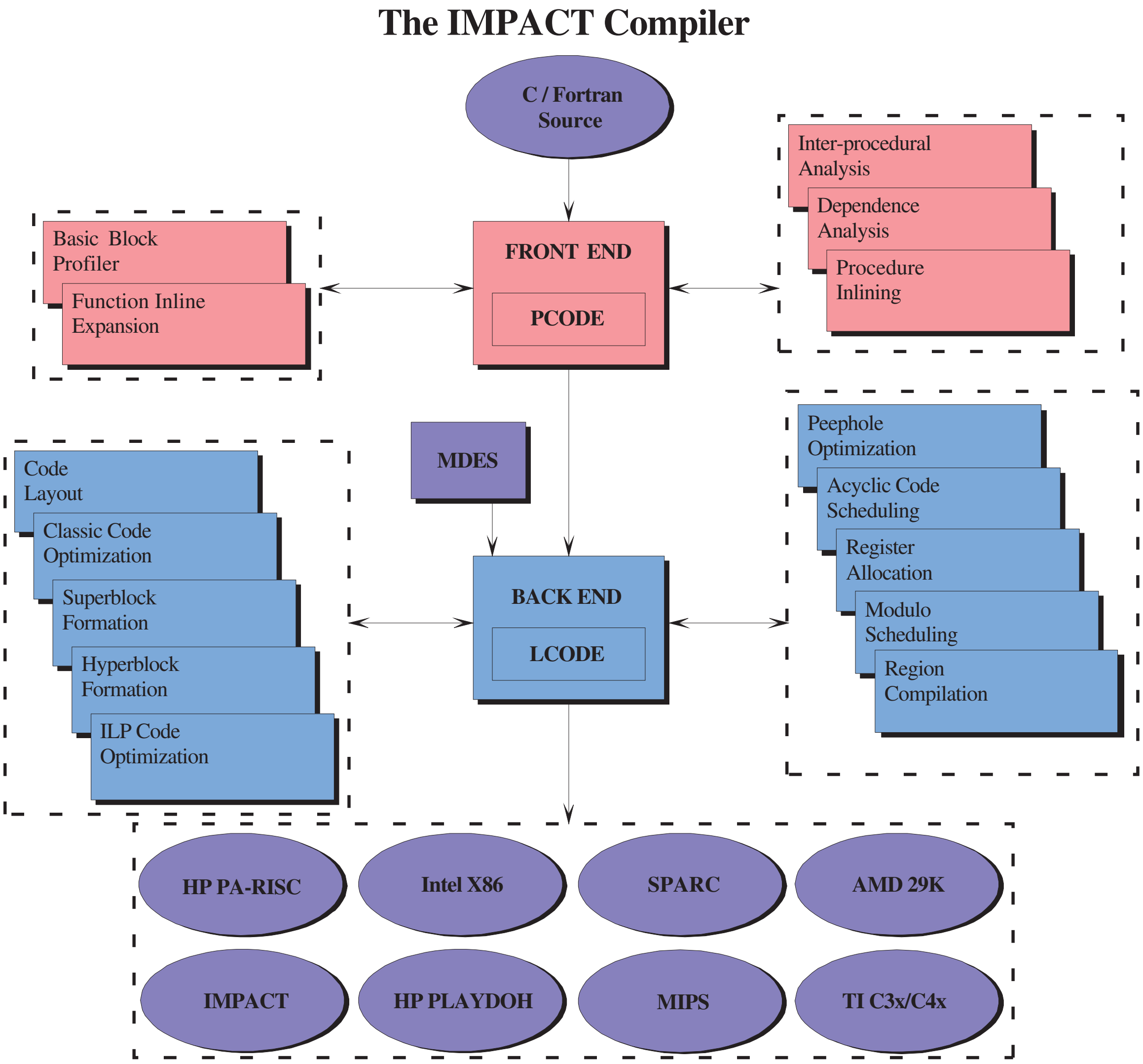

Figure 2.1: The IMPACT compiler

The machine independent nature of the *Lcode* format has facilitated the creation of several code generators for several different architectures [7]. The most actively supported architectures are the Sun SPARC, the HP PA-RISC, and the Intel X86. The two main components of code generation are the instruction scheduler and the register allocator [15]. Several scheduling models exist, including acyclic global scheduling [16], [17], sentinel scheduling [18], and software pipelining using modulo scheduling [19].

The IMPACT and HPL Playdoh [20] architectures, two experimental instruction-level parallelism (ILP) architectures, are also supported. These experimental architectures provide the necessary framework for advanced compiler and architecture research. In fact, the specifications for the architectures are based on parameterizable resources that allow the exploration of a wide variety of machines. The IMPACT compiler meets these specifications by using the technology of a machine description database, *Mdes* [21]. The *Mdes* contains a large set of information to assist optimization, scheduling, register allocation, and code generation. Information provided by the *Mdes* includes the number and type of available function units, size and width of the register file, instruction latencies, instruction operand constraints, addressing modes, and pipeline constraints. In addition, both architectures support forms of speculative and predicated execution. For this thesis, all experiments utilize the IMPACT architecture.

2.2 Overview of *Lcode* Structure

As with any intermediate representation, each *Lcode* function consists of a hierarchy of three data structures: a function block, control blocks, and operations. An operation is a fundamental element of computation. A control block is a set of operations with the characteristic of a single entry point and multiple exit points. A control block can also be referred to as a basic block when only one exit point exists. A function block is made up of a sequence of control blocks as shown in Figure 2.2. Each data structure in the *Lcode* function hierarchy is marked with a representative keyword. A function block is marked with the keyword *function*, control blocks with *cb*, and operations with *op*. Following each key word is an identifier, which gives the function, control block, or operation a unique identification. For functions, the identifier is the function name, and each control block and operation is given a unique numerical identification within that function. Different phases of compilation can change the *Lcode* instructions (or operations).

Lcode instructions are described in the three address notation [22]. *Lcode* is an instruction set for a load-store architecture, supporting an unlimited number of virtual registers and basic synchronization instructions. It is broken down into data and function blocks. Each *Lcode* instruction is composed of the following four major parts: an operation number (identification), an opcode, operands, and attributes, as shown in Figure 2.2. In addition, a predicate register input and flag attributes can be specified on an instruction. The opcode signifies which operation is to be performed. The available operations can be broken down into groups of arithmetic and logic unit (ALU), memory access, and

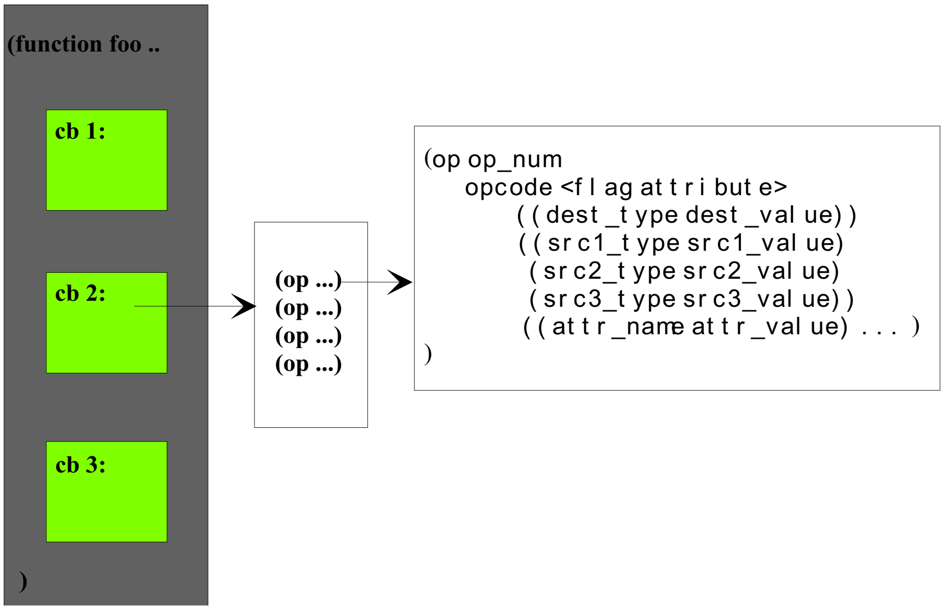

Figure 2.2: *Lcode* function block layout

control flow operations. A detailed description of the *Lcode* instruction requirements can be found in [23].

3. EMULATION ENVIRONMENT

The IMPACT simulation environment provides the necessary facility to evaluate the effects of compiler techniques on various research architectures. It also functions as an environment used for profiling the control flow of a program. One of the major goals of this thesis is to make the profiling aspect of the simulation environment portable to various platforms. The following sections provide an overview of the current environment and the alternative as presented by this thesis.

3.1 Overview of the IMPACT Simulation Environment

The simulation environment involves three important technologies: emulation, probe insertion, and a trace-driven performance simulator. Figure 3.1 presents a block diagram of this environment, which describes the interconnection of the three technologies. Emulation provides a program with the appearance of the architectural functions which are not directly implemented in the target machine. For instance, predicated execution must be emulated for the IMPACT *Lcode* because current processor technology does not include predication support. Other examples of emulated features are speculative execution, very long instruction word (VLIW) execution semantics, rotating registers, memory conflict buffer (MCB) functionality, or even entire instruction set architectures (ISA) support.

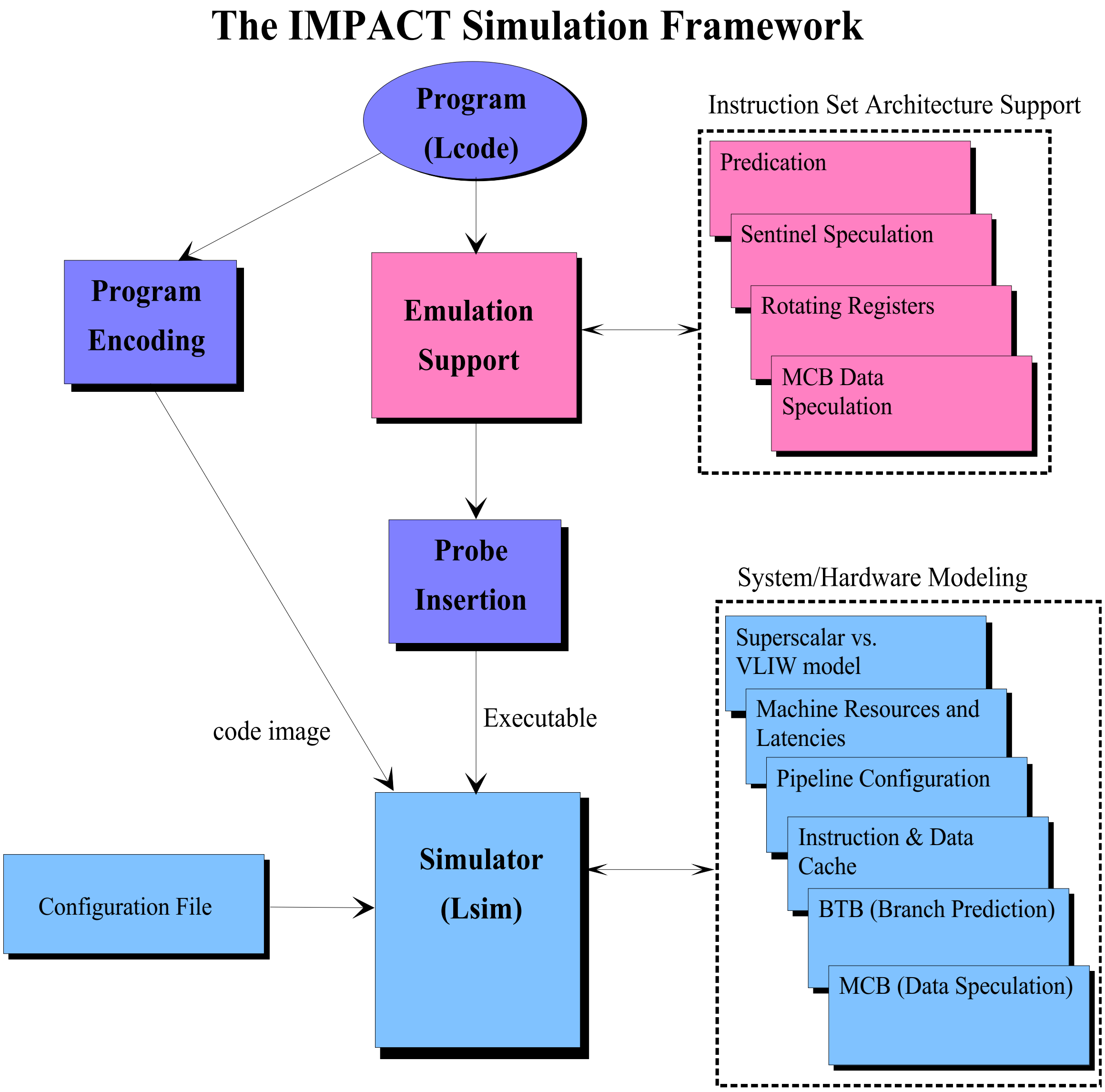

Figure 3.1: The IMPACT simulation framework

Probe insertion into program code allows trace information to be generated while running the executable. Trace information consists of memory target addresses, branch direction, predicate value generations, and jump target addresses. The IMPACT simulator, *Lsim*, is a trace-driven framework which is capable of cycle-by-cycle simulation of cache memories, branch prediction hardware, instruction pipelines, superscalar and VLIW processors, and MCB hardware. *Lsim* processes trace information and coordinates it with an encoded version of the *Lcode* files. This enables it to coordinate program references with simulated machine structures. For instance, trace information containing branch directions is used by *Lsim* to update simulated branch target buffer (BTB) structures as well as follow the proper path of execution. This coordination allows *Lsim* to attribute variable machine cycles to the operations implied by the trace information. In short, the probed executable provides the functionality of the program being simulated, while *Lsim* estimates the target machine cycles.

3.2 Overview of the *Lcode* Emulation Environment

Profiling is a technique for instrumenting programs in order to collect run-time information. control-flow and memory-dependence are the two most common types of profiling that are used to guide optimizations. Control-flow profiling collects information about the relative frequency of execution paths. Memory-dependence profiling summarizes the frequency of address conflicts between two memory references.

For the purpose of aggressive instruction scheduling, control-flow profiling provides the following information: an invocation count of each function, an execution count of each control block, and the branch taken frequency of two-way and multi-way branches. The current IMPACT methodology inserts probes into the program intermediate representation and links the probed program with an architecture-specific probing library to form an executable. Each of the existing code generators supported in IMPACT require individual profiling libraries formed using custom assembly functions. Such libraries complicate and extend the development time of code generators. When the executable is run in conjunction with the simulator, profile information is collected for the user specified input.

Figure 3.2 presents a block diagram of the profiling implementation being presented in this thesis that avoids the issue of providing architecture-specific probing libraries. Since the program is being emulated in C, no architectural knowledge is required for instrumenting the program. In the IMPACT methodology, instrumentation involves several steps of inserting probing instructions before and after register allocation of a program. Similarly, in this implementation, instrumentation refers to inserting probing C statements during the translation process. These probing C statements are profile-gathering function calls, which are inserted where appropriate to increment profile counters. The profile-gathering functions inserted in the body of the functions are used to indicate the following: function entry, function reentry from a subroutine call, control block entry, and branch or hashing jump behavior.

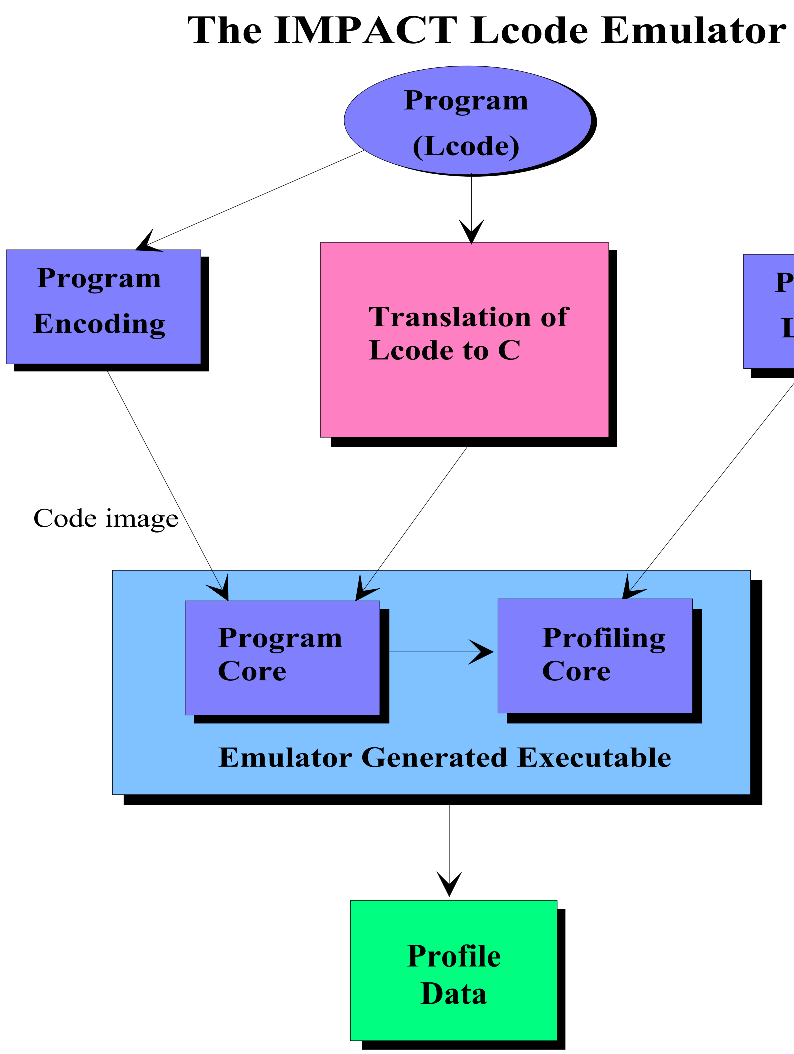

Figure 3.2: The *Lcode* emulation framework

There are two main data structures that serve as the foundation for collecting profile information in this environment. The data structures used to collect the execution profile of each function are shown in Figures 3.3 and 3.4 . These structures and profiling functions are part of the architecture independent library that is linked in for profiling purposes. Upon execution of the program, in this profiling implementation, an encoded image of the program is loaded to dynamically allocate space for all of the required profile counters. After building the counters, the program continues with these counters recording execution profile statistics.

```
typedef struct profile{
  char *name;                /*function name*/
  int func_weight;           /*count of function entry*/
  int max_branch;            /*number of branches/jumps in this function*/
  int *br_jmp_executed;      /*execution count of branches/jumps*/
  int *br_jmp_taken;         /*number of times branches were taken*/
  char *br_jmp_type;         /*cond. branch/pred jump/jump/hashing jump*/
  int num_hash_jumps;        /*number of hashing jumps in this function*/
  HASHPROF *hjmp_profiles;   /*profiles for hashing jumps*/
  int max_cb;                /*max control block id*/
  int *cb_order;             /*sequential order of cb's in file*/
  int *cb_profiles;          /*count of control block entry*/
  struct profile *next_profile;
} PROFILE;
```

Figure 3.3: *Lcode* emulator internal representation: PROFILE structure

```
typedef struct hashprof{
  int num_conds;           /*number of possible branching condition*/
  int *cond;               /*array of conditions*/
  int *weight;             /*taken weight based on condition*/
} HASHPROF;
```

Figure 3.4: *Lcode* emulator internal representation: HASH_PROF structure

This approach is generally observed as being faster than the current method of profiling using the simulator. For example, the profiling process is approximately four times faster on the benchmark program *134.perl* using the test input set. Overall the benchmark programs, this approach is 38 % faster than using the IMPACT simulator. The dynamic allocation approach is used in the *Lcode* emulator because it is relatively fast and can handle most programs. The speed-up ratio depends on size of the input set and the program. Finally, the profile information collected is mapped into the original intermediate code (*Lcode*) using the function names and the control block identifiers.

4. BENEFITS OF EMULATING THE INTERMEDIATE REPRESENTATION

One of the primary goals in the development of the *Lcode* emulator is to provide the IMPACT compiler with a framework for cross-platform development. The IMPACT compiler currently has some utilities that enable it to exhibit this characteristic, however, it is limited by the need to provide several architectural specific tools. Thus, this chapter presents the *Lcode* emulator's approach to cross-platform development. In addition, the additional debugging capability provided by emulating the intermediate representation in a high-level language is also discussed.

4.1 Cross-Platform Development

As shown in Figure 2.1, a C program goes through several phases of translation to intermediate representations which facilitate code optimizations. Finally, the intermediate representation serves as the input to an architecture-specific code generator, which translates it to the actual machine language. Since the IMPACT compiler is heavily dependent on profile information for guiding code optimizations, this creates a situation in which a code generator and a probing library are required for every development platform.

Consider the following scenario. A developer would like to compile code for a TI C3x/C4x digital signal processing processor using the IMPACT compiler. The I/O capabilities of this processor are very limited and insufficient for gathering profile information. Also, due to the embedded nature of this processor, the memory requirement for running the compiler is not adequate. Since a code generator exists for this architecture in the IMPACT compiler, it makes producing an executable for it trivial. However, in order to produce the highly optimized code, a different platform is required for profiling and simulation because of the limited resources of the embedded processor. Assuming the development platform is an X86 machine, this would demand an additional code generator for the development platform. In addition, a probing library specific to the X86 architecture becomes a critical part of the development process. In order to remove the burden or cost of providing these extra utilities and generate very efficient code, it becomes paramount to provide a developer with the means to compile code efficiently for a target architecture on a different development architecture.

The profiling approach presented in this thesis provides this cross-platform development capability. In this approach, generating profiling information involves running the *Lcode* emulator on the *Lcode* to generate a profile-gathering oriented executable. Since the emulation is in C, it can be compiled using the existing software tools on any system. Running the executable generates the profile information in a format that can be mapped back into the *Lcode* for further optimizations by the IMPACT compiler. This approach essentially abstracts away the notion of distinct development and target architecures in the effort to generate efficient code.

4.2 Debugging Facility

Emulation of *Lcode* in C provides a developer with a vast array of tools with which to verify and debug the intermediate representation. One of these tools is a C source level debugger, e.g., GNU debugger (gdb) [24]. Upon compilation of one or more C source files, a program suitable for execution is created. The executable program's symbol table contains information used by the loader for creating its process image, for resolving external references, and debugging. With the information contained in an executable, a typical debugger allows a developer to inspect a program as it executes. The most common debugger is a breakpoint debugger. Inspection of a program is possible because these types of debuggers can suspend the program at a particular point. In addition to inspection of the program state, modification is also possible.

Program suspension is implemented by inserting breakpoints in the body of the program. Breakpoints are implemented by replacing the instruction at the desired location with a hardware trap instruction. In addition, the debugger stops the program execution if any exception is raised during the program's execution. To continue the execution, the program proceeds with the real instruction that was replaced by the trap instruction. Continuation can be implemented in two different ways, either the real instruction is emulated, or the trap instruction is replaced with the real instruction. In the latter case, the debugger is single stepped, and the trap instruction is eventually put back in place of the real instruction. Program execution can proceed until completion, or the next breakpoint.

The benefit of having a generic C source level debugger is that a code generator is not required to debug a program. Currently, the only means of debugging code generated by the IMPACT compiler is to use an a architectural-specific debugger on the assembly source code generated. Debugging at this level can be quite tedious, and it requires understanding the semantics of the development platform's assembly language. As a consequence, it is not always easy to establish a one-to-one correspondence between the intermediate representation and the assembly source code. The *Lcode* emulator provides a couple of debugging aids to alleviate these problems. The emulator can generate C source code with the operation number of each *Lcode* instruction placed on the same line as its equivalent C statement. In addition, a file containing a line-to-line mapping between a *Lcode* file and it corresponding C file can be generated to further aid the debugging process.

5. IMPLEMENTATION DETAILS

The *Lcode* emulator's primary function is to convert the IMPACT intermediate representation into an equivalent form in the C programming language. The emulation of a *Lcode* function in C proceeds in two phases. The first phase performs all the initialization of the emulator's internal data structures with the function's information. The second phase then processes these internal data structures and produces an equivalent C function as its output. In additon to *Lcode* functions, there is a global data section which is also processed in the two-pass manner described above. These phases of the translation process and some of the internal data structures are described in the following sections. Finally, an overview of the emulation support provided for predicated and speculative instructions is presented.

5.1 Initialization of Internal Data Structures

This phase involves the gathering of information from the *Lcode* structures and filling in the appropriate emulator structures. One of the first characteristics determined in this phase is the kind of function being processed; i.e., is it a function with a fixed number of arguments or a variable argument function? The number and types of the function arguments are determined and inserted into a dynamic list. The distinction in the kind of functions and other relevant issues will be examined in section 5.2.1.

Since the *Lcode* format utilizes virtual registers, essentially an infinite number of registers, the *Lcode* emulator determines the maximum register identification number (id) and uses it to dynamically allocate space for its internal data structure. This space is used to record the characteristics of these registers. In most cases, many of the register numbers between zero and the maximum register id are not used. Thus, all the register numbers are remapped to a new identifier in the *Lcode* emulator. The importance of this remapping is that it reduces the amount of statically allocated space for local variables in the C source code generated.

After these initial determinations, each instruction in the function is further processed to complete the initialization phase. In the *Lcode* format, the data type of each operand is specified for each instruction. The data type of a register operand can be one of the following: an integer, a float, a double, or a predicate. As shown in Figure 5.1, the destination register of *op* 8 is the integer register number 9. In the *Lcode* emulator, data types such as *char*, *short*, etc., are promoted to an integer type using C's casting operator. These data types are used to set the register type in the *Lcode* emulator's internal data structures.

The next step involves determining if any data labels exist in the instruction. In the IMPACT compiler, all data labels are global variables defined in a data block. Since the data block is expected to be in a file exculsively, all the data labels can be declared as external variables. External data variable declarations are expected to be outside and before the function definition in the C source file. Thus, a linked list of external data labels is also built in this phase.

Another major part of this phase is the processing of arguments to subroutine calls. In the IMPACT arhitecture, the first four arguments of a subroutine are passed via register; any extra arguments are passed via memory. The callee function can access arguments passed via memory from its incoming parameter space, i.e., the callers outgoing parameter space. However, the *Lcode* emulator passes these via-memory arguments as regular function arguments through the standard C calling convention. Thus, it becomes necessary to keep track of how many extra local variables are required for passing via-memory arguments. Section 5.2.4 describes the parameter space in detail.

```
(cb 1 1.000000 [(flow 0 2 1.000000)] )
  (op 1 define [(mac $P0 i)] [])
  (op 2 define [(mac $P1 i)] [])
  (op 3 define [(mac $return_type i)] [])
  (op 4 define [(mac $local i)] [(i 16)])
  (op 5 define [(mac $param i)] [(i 16)])
  (op 6 define [(mac $swap i)] [(i 104)])
  (op 7 prologue [] [])
  (op 8 mov [(r 9 i)] [(mac $P0 i)])
  (op 9 mov [(r 10 i)] [(mac $P1 i)])
```

Figure 5.1: The first control block of a typical function in the *Lcode* format

The part of a *Lcode* function that has the most impact on the C source code produced is the first control block. Figure 5.1 is an example of a typical first control block. The formal arguments to the function are specified as indicated by *op* 1 and *op* 2. The function return type is given by *op* 3. Next, *op* 4 indicates the amount of bytes required for the local stack; in this example, an allocation of 16 bytes is required. Similarly, *op* 5 and *op* 6 specify the amount of bytes required for the out-going parameter space and the swap space respectively. The out-going parameter space represents one of the common ways of communicating parameter passing in intermediate representations. The swap space requirement appears only if the *Lcode* function has gone through the IMPACT register allocator [15]. This space is used for register spilling when there are not enough physical architectural registers.

5.2 Conversion of *Lcode* to C

This phase uses all the information in the *Lcode* data structures and the emulator's data structures to transform *Lcode* instructions into C statements. Since all of the functions generally do not reside in the same file, a header file consisting of extern declarations of all non-library functions is produced. This header file is included by all the C source files generated. In addition, a special library header file is included for variable argument functions. As mentioned earlier, *Lcode* is broken into data and function blocks. All data blocks (i.e., global variables) with the exception of hashing jump data blocks reside in one file, while all the function blocks can reside in one or more files. This arrangement makes all data variables global, hence requiring external declarations in the C function files generated. The following sections describe in detail the major aspects involved in the transformation from *Lcode* instructions to C statements.

5.2.1 Function declaration

As shown in Figure 5.2(c), the function definition uses the Kernighan and Ritchie (K&R) standard [6] for argument declaration. This stems from a couple of reasons: function prototyping and automatic argument conversions. The *Lcode* emulator generates a header file which contains an external declaration for all the functions in the program. These function prototypes can be referred to as function allusion because the types of the formal arguments of the functions are not specified (see Figure 5.2(b)).

```
(global _main)
(function _main 1.000000
 (cb 1 1.000000 [(flow 0 2 1.000000)])
   (op 1 define [(mac $P0 i)] [])
   (op 2 define [(mac $P1 i)] [])
   (op 3 define [(mac $return_type i)] [])
   (op 4 define [(mac $local i)] [(i 0)])
   (op 5 define [(mac $param i)] [(i 16)])
   (op 6 prologue [] [])
   (op 7 mov [(r 9 i)] [(mac $P0 i)])
   (op 8 mov [(r 10 i)] [(mac $P1 i)])
   ....
   (op 108 jsr <E> [] [(l _$fn_wcp)]
   <(tr (mac $P0 i)(mac $P1 i)(mac $P2 i)
   (mac$P3 i))(ret (mac $P15 i))>)

                (a)
```

```
#ifndef _EXTERN_H
#define _EXTERN_H
extern int wcp();
extern int ipr();
#endif /* _EXTERN_H */
       (b)
```

```
int  main(P0, P1)
int P0;
int P1;
{
 ...
 P15 = wcp(P0,P1,P2,P3);
 ...
}
       (c)
```

Figure 5.2: (a) Original *Lcode*, (b) header file, and (c) function declaration

Function allusions are used in place of ANSI standard [6] function prototypes in order to minimize the effort required to build the header file. Arguments are automatically converted to a different type when function allusions are used. The data types *chars* and *shorts* are converted to *ints*; *floats* are converted to *doubles*. If the combination of function allusions and the ANSI style of function declarations are used, erroneous results may be produced due to incorrect argument conversions. Consider the functions *foo* and *foo_bar* shown in Figure 5.3. Assuming the function definitions reside in different files, an extern declaration is required for *foo_bar*.

```
File_1:
extern foo_bar();
foo(int xtra){
   float var1,var2;
   var1 = 0.0,var2 = 0.0;
   var1 = foo_bar(var1, var2);
}
```

```
File_2:

float foo_bar(float x, float y){
      float z;
      z = x + y;
      return z;
 }
```

Figure 5.3: Illustration of automatic argument conversion problem

As shown in the definition of *foo_bar*, it requires that its arguments be of type *float*. However, when the C compiler generates assembly code for the function *foo*, variables a and b are promoted from *floats* to *doubles* in accordance with C conventions. Since the ANSI style is used for *foo_bar*, the assembly code generated by the compiler does not include instructions to convert the argument from *doubles* to *floats*. If the K&R style of function declaration is used, the C compiler inserts instructions to convert the arguments passed to *foo_bar* to their correct data types. The only exception to this approach of function declaration and prototyping is the variable argument function. The

ANSI standard is used in this instance, in order for a C compiler to correctly handle subroutine calls to a variable argument function with varying numbers of arguments within the same caller function. For a function with a fixed number of arguments, all the incoming arguments are specified in the first control block of the *Lcode* function. These arguments are either passed through registers or memory. Figure 5.2(a) shows an example of a fixed argument function in which all the arguments are passed via registers. The other type of function is the variable argument function. These functions have an attribute named VARARG attached to the function declaration in the *Lcode* format as shown in Figure 5.4. The attribute also includes an integer value that reflects the characteristics of the argument list. The VARARG attribute value is zero or less, since the IMPACT stack implementation uses negative addressing. The value primarily specifies the offset to the first unnamed argument in the argument list.

```
(function _xlsave 0.000000
          <(VARARG (i -8))>)
  (cb 1 0.000000  [(flow 0 2 0.000000)])
    (op 1 define [(mac $P0 i)] [])
    (op 2 define [(mac $P1 i)] [])
    (op 3 define [(mac $local i)] [(i 0)])
```

```
#include <stdarg.h>
int  xlsave (int P0, ...)
{  int P1;
   va_list ap;
   va_start(ap, P0);
   ...
```

Figure 5.4: *Lcode* and C variable argument function declaration

In C, there are two ways of declaring a variable argument function. The first method uses the ANSI standard declaration as shown in Figure 5.4. This method can accept variable arguments in a portable fashion. Without conforming to the ANSI standard, writing a variable argument function would require knowledge about the stack implementation of a particular compiler. The header file, *stdarg.h*, is included in the source file for the ANSI style declaration. The second method does not conform to the ANSI standard and requires the inclusion of the header file, *varargs.h*. One reason why this method is not very portable is that the definition of the symbolic name *va_dcl* varies based on the header file on a system. Figure 5.5 shows an example of this type of function in its original C and the resulting *Lcode* form. The VARARG attribute value is equal to zero for this function since the whole argument list is considered to be variable.

```
#include <vararg.h>
void pm_message(va_alist)
va_dcl
{
     ...
```

```
(function _pm_message 0.000000
          <(VARARG (i 0))>)
  (cb 1 0.000000 [(flow 0 2 0.000000)])
    (op 1 define [(mac $P0 i)] [])
    (op 2 define [(mac $local i)] [(i 0)])
```

Figure 5.5: Non-ANSI style and *Lcode* variable argument function declaration

In translating the declaration of variable argument functions, the *Lcode* emulator ignores the offset value on the VARARG attribute and just notes the variable argument nature as mentioned in section 5.1. All variable argument functions are translated to an ANSI style function regardless of their original C source declarations. If a function has more than one argument, this is an indication that the original C source code conformed to the ANSI standard. Hence, the *Lcode* emulator replaces the last argument in the argument list with the C ellipsis notation (...), as shown in Figure 5.4. Although *P1* is an argument to the function in *Lcode* representing the variable part of the argument list, it is declared as a local variable in C due to its replacement with the ellipsis. In addition, *Lcode* emulator also notes the last named argument, in this case *P0*. This is required for calls to the C macro *va_start*. If the function declaration has only one argument, then the original C source code did not conform to the ANSI standard. In this case, the single argument is used as the last named argument to *va_start* macro call. In the C source code generated, the ellipsis notation is also included in the declaration as shown in Figure 5.4. The only difference in the definition of *xlsave* and *pm_message* is the variable *P1*. Since *P1* is not an argument to *pm_message*, it is not declared as a local variable.

After the function declaration is completed, the *Lcode* emulator inserts a declaration for the local variable *ap*, argument pointer, of type *va_list*. The type *va_list* defines an array type suitable for refering to each argument in turn. The macro *va_start* initializes *ap* to point to the first unnamed argument. This macro must be called once before the variable *ap* is used. The *Lcode* subroutine, *__builtin_va_start*, is replaced with the *va_start* macro by the *Lcode* emulator.

5.2.2 Local data and variable space

Another important step in the emulation process is the transformation of register usage in *Lcode* to local variables. Registers can be emulated by declaring an array of integers, floats, or doubles as specified by the maximum register id. However, since all the available register ids are not actually used, a register remapping function creates new maximum register ids based on the data types used in the function. This, in effect, reduces the number of useless and wasteful local variable declarations. If the *Lcode* includes predicate instructions, an integer array is used to emulate the predicate registers. While only one bit of these variables is used, the run-time overhead of using a single integer variable to represent 32 predicate registers would be too exorbitant (i.e., bit-wise operations). A declaration of an array of integers is also used to allocate the local stack, swap, and out-going parameter space.

5.2.3 Conditional branches, jumps, and hashing jumps

Control block identifiers are converted to labels in the C source code that is generated. This implementation is straightforward and enables the use of C *goto* statements to emulate jumps and conditional branches. Another peculiar transformation of *Lcode* instructions involves hashing jumps, which are the result of switch statements in the original C source code. The general form of a hashing jump is shown in Figure 5.6.

```
(cb 16 0.000000
     [(flow 21 5 0.000000)
      (flow 27 8 0.000000)
      (flow 28 7 0.000000)
      (flow 26 7 0.000000)
      (flow 29 5 0.000000)
      (flow 31 8 0.000000)
      (flow 33 9 0.000000)
      (flow 2147483647 3 0.000000)])
(op 27 blt [][(r 3 i)(i 21)(cb 3)])
(op 28 bgt [][(r 3 i)(i 33)(cb 3)])
(op 29 ld_i [(r 17 i)][(r 84 i)(l hash)]
     <(label (l hash))(param (i 0))>)
(op 30 jump_rg [][(r 17 i)(mac $P0 i)])
```

(a)

```
(ms data)
(align 4 hash_0)
(reserve 64)
(wi (l hash)(l cb5_x))
(wi (add (l hash)(i 4))(l cb5_x))
(wi (add (l hash)(i 8))(l cb3_x))
(wi (add (l hash)(i 12))(l cb3_x))
(wi  ...)
(wi (add (l hash)(i 36))(l cb8_x))
(wi (add (l hash)(i 40))(l cb7_x))
(wi (add (l hash)(i 48))(l cb3_x))
(wi (add (l hash)(i 52))(l cb8_x))
(wi (add (l hash)(i 56))(l cb3_x))
(wi (add (l hash)(i 60))(l cb9_x))
```

(b)

Figure 5.6: (a) Hashing jump usage, and (b) data section for hashing jump

This conversion is based on profile information and is intended to reduce the amount of comparision required before reaching the appropriate block of code. As shown in Figure 5.6, an index into the jump table is used to load the id of the target control block. In addition, the control block also has a series of *flow_arc* attributes attached to it. The *flow_arcs* are used to describe the control-flow in the current control block. Each *flow_arc* contains the keyword *flow*, the branch condition value, target control block, and the taken frequency. These *flow_arcs* are essential to the transformation of a hashing jump into C statements. Since the *flow_arcs* contains information about the possible paths that can be taken from the control block, they are converted to C if-statements as shown in Figure 5.7 and are used in place of the hashing jump.

```
CB_16:   if (IR[3] < 218) goto CB_3;
         if (IR[3] > 233) goto CB_3;
         /* switch/hashing_jump statements */
         if (P0 == 218) goto CB_5;
         else if (P0 == 227) goto CB_8;
         else if (P0 == 228) goto CB_7;
         else if (P0 == 226) goto CB_7;
         else if (P0 == 219) goto CB_5;
         else if (P0 == 231) goto CB_8;
         else if (P0 == 233) goto CB_9;
         else goto CB_3;
```

Figure 5.7: Emulation of a hashing jump in C

5.2.4 In-coming and out-going parameter space

Perhaps one of the most important aspects of the *Lcode* is the parameter space used for variable passing between functions. Figure 5.8 shows the general structure of parameter passing between functions in the *Lcode* format.

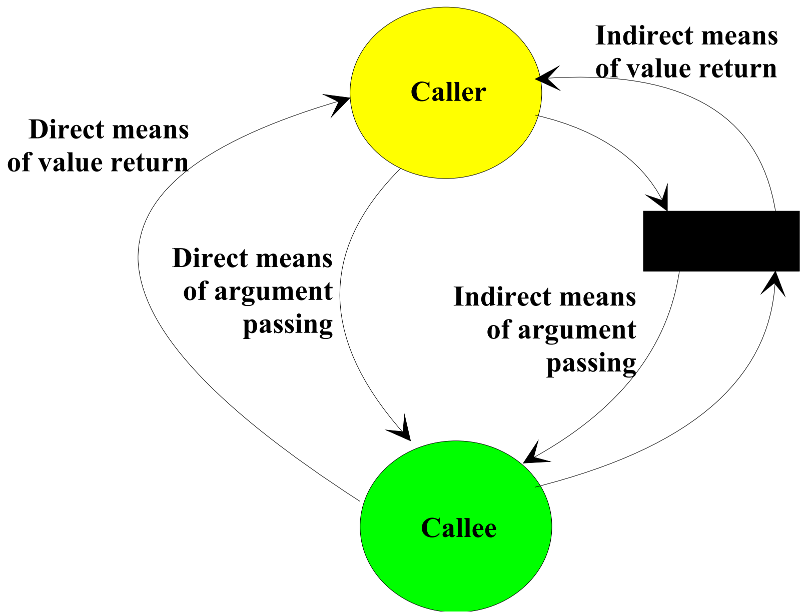

Figure 5.8: The general convention between caller and callee functions in *Lcode*

In the IMPACT architecture, only four registers are reserved for argument passing between functions. Thus, functions with more than four arguments have their remaining arguments passed through memory. The parameter space of a caller function can be referred to as the in-coming parameter space of a callee function. These spaces are pointed to by IP and OP in the *Lcode* format. Figure 5.9 illustrates the structure of a function in the IMPACT architecture. In addition, it also shows how this structure is emulated in C. A global integer pointer, *global_OP_ptr*, points to the current out-going parameter space. Each function has a local pointer called *previous_OP_ptr*. At the start of each function, the *previous_OP_ptr* is set to point to the *global_OP_ptr* (i.e., the outgoing parameter space of the caller function). Hence, the *previous_OP_ptr* emulates the IP pointer in the *Lcode* format. The *global_OP_ptr* is now set to point to the local parameter space (i.e. this function's outgoing parameter space). Before each return statement in the

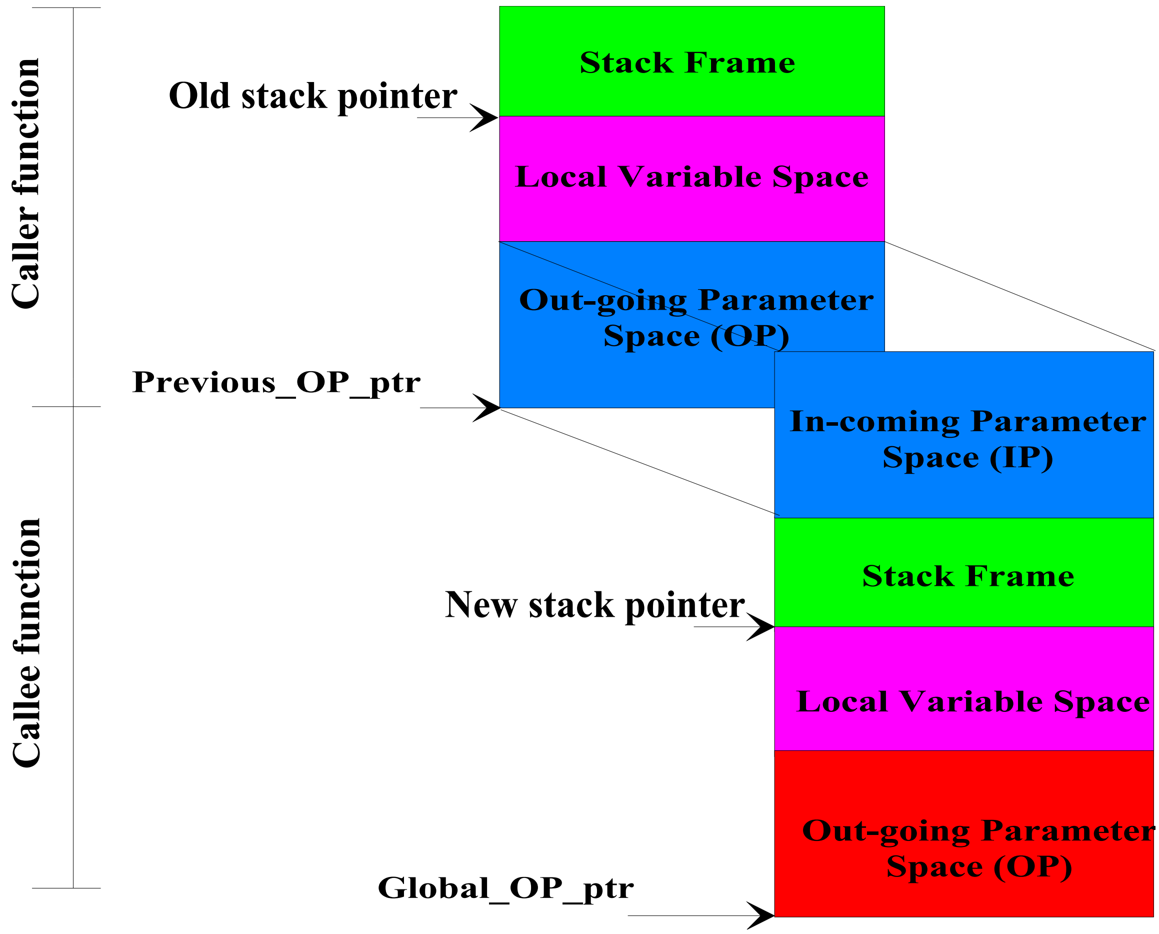

Figure 5.9: The structure of an *Lcode* function

C source code, the *global_OP_ptr* is reassigned to its previous value (the *previous_OP_ptr* in the callee function). This implementation allows the chaining of parameter spaces involved in a caller-callee series.

5.2.5 Global data declaration

As mentioned earlier in the chapter, the *Lcode* emulator expects the data block of a program to reside in a file without any function blocks. The data block can be further classfied into initialized and uninitialized data sections. The *Lcode* operation, *ms type*, is used to signify the beginning of a new data section. The keyword *ms* stands for *memory segment*. The value for *type* is either *data* for initialized sections or block starting space, *bss*, for an uninitialized sections. The declaration of global data variables consist of several operations. These *Lcode* data operations can be classified into five groups. A complete description of each group can be found in [23]. The following is a brief overview of the operation groups:

Group 1 (data_op value): Currently *data_op* can only be *reserve*. It is used to indicate that a memory block of *value* number of bytes should be reserved.

Group 2 (data_op name): The value of *data_op* is either *void* or *global*. This operations is used to introduce a new data variable, *name*. The *global* keyword is used to promote the scope of the variable.

Group 3 (type size name expr*): This operation is used to initialize the data variable *name*. The *type* can take on values of *byte, word, long, float,* or *double*. The *size* defines the number of units of *type* to allocate. The initialization value is computed by zero or more *expr* in the operation. Specifically, this operation is used to initialize scalar data types.

Group 4 (type $\mathbf{expr}_1$ $\mathbf{expr}_2$): Typically, the initialization of aggregate data types, such as arrays and structures, utilizes this operation. It can have *type* values of *wb, ww, wi, wf, wf2,* or *ws.* The size of these fields are *byte, word, integer, float, double,* and *string pointer*, respectively. The operations writes the computation result of $expr_2$ to the address determined by the computation result of $expr_1$.

Group 5 (data_op value name): For this group, *data_op* can only have values of *align* or *element_size.* The *align* keyword is used to define the appropriate alignment for an allocated memory block. While, the *element_size* indicates the size of elements in an aggregate data type. The *value* specifies in number of bytes the alignment size or the element size.

Several combinations of the operation groups are used for a single variable declaration. The declaration of a scalar variable is shown in Figure 5.10(a). This declaration is made up of a group 1 and a group 3 operation. The equivalent C statement generated by the *Lcode* emulator for the variable *ratname* is shown in Figure 5.10(b). In *Lcode* data declaration, if the expression part of a group 3 operation is nonexistent as is the case for the *casecount* variable, the C declaration is of the form of an uninitialized scalar variable as shown in Figure 5.10(c).

```
(global _ratname)
(long 1 _ratname (s ``./nrform''))    char *ratname = ``./nrform'';
(global _casecount)                                (b)
(long 1 _casecount)                   int casecount;
         (a)                                       (c)
```

Figure 5.10: (a) *Lcode* scalar variable, (b) equivalent C declaration, and (c) uninitialized C declaration

Aggregate data types generally combine operations from groups 1, 2, 4, and 5. As mentioned earlier, group 4 operations are used solely for the initialization of these data structures. If an aggregate data type is uninitialized, the *Lcode* emulator translates all its memory allocation into a C character array declaration for the number of bytes reserved. Thus, an uninitialized *integer, float,* or *double* array would have the same type of C declaration. An example of uninitialized memory allocation is shown in Figure 5.11. This is sufficient for the correct functionality since the casting operator is used when loading or storing data to the allocated space. In this declaration, the correct variable type is either a *float* or an *integer* based on the *align* operation. However, this is not sufficient for determining the correct type.

```
    (ms bss)
    (global _buf)
    (align 4 _buf)                        char buf[520];
    (element_size 4 _buf)
    (reserve 520)
         (a)                                   (b)
```

Figure 5.11: (a) *Lcode* aggregate variable, and (b) equivalent C declaration

If the aggregate data type is initialized, the alignment directive is used to determine the data type. In addition, the initialization fields of an aggregate data type can be used to distinguish between an array and a structure. Generally, if the *element_size* value differs from the *align* value, the *Lcode* emulator assumes the variable is a structure. Figure 5.12 shows an example of an array declaration in the *Lcode* format and its equivalent C declaration. The declaration for an array of structures is similar to the simple array

```
(global _big)
(align 4 _big)
(element_size 4 _big)
(reserve 12)
(wf (l _big) (f 0.00000000e+00))
(wf (add (l _big)(i 4)) (f 1.00000001e-01))
(wf (add (l _big)(i 8)) (f 2.00000003e-01))
```
(a)

```
float big[3] = {
0.000000,
0.100000,
0.200000
};
```
(b)

Figure 5.12: (a) *Lcode* array declaration, and (b) equivalent C declaration

declaration shown in this example. The main difference is that the *element_size* value differs from the *align* value and is typically greater than eight (i.e, a *double* data type requires eight bytes). The *Lcode* emulator translates this type of structure by flattening it to take the form of a simple structure declaration. Figure 5.13(a) shows the *Lcode* declaration for a simple structure variable; it is worth noting that the *element_size* operation is not part of the declaration. Finally, if the *type* field of the *group 4* operations used for the initialization are not uniform, then this is a clear indication that the variable is a structure.

```
(global _jtype)
(align 4 _jtype)                                    struct jtype_type {
(reserve 12)                                               int var_0;
(wi (l _jtype) (i 65))                                     int var_1;
(wi (add (l _jtype)(i 4)) (i 3))                           float var_2;
(wf (add (l _jtype)(i 8)) (f 4.44999993e-01))  } jtype = { 65, 3, 0.445000 };
          (a)                                                (b)
```

Figure 5.13: (a) *Lcode* structure declaration, and (b) equivalent C declaration

5.3 Predication and Speculation Support

Up to this point, the functionality provided by the *Lcode* emulator has covered features that are common place in modern processors. This section describes the ability of the *Lcode* emulator to handle novel architectural features that are being introduced in modern processor. The increasing ability of the modern processor to execute multiple instructions in a single cycle vastly impacts the performance of these machines. The primary means of meeting this requirement is to expose a large amount of instruction level parallelism. However, this requires a means of overcoming the limitations imposed by branch instructions. Branches impose control flow dependencies, which forces the compiler to make conservative decisions. In order to alleviate these artificial performance bounds imposed by branches, an effective means of eliminating branches from the instruction stream is required.

5.3.1 Predication

Predication [25] has been shown to be an architectural feature that exposes ILP in a typical application program. Predication presents an elegant solution for converting program control flow into data flow. The essence of predicated execution is the ability to suppress the modification of the processor state based upon some condition. The condition of the branch is used to set Boolean operands, which guards the execution of instructions in the branch's paths. This feature allows instruction from both paths to be fetched and executed simultaneously. The result of the condition which determines if an instruction should modify state is stored in a set of 1-bit registers. These registers are collectively referred to as the predicate register file. Instructions that operate on the predicate registers can be grouped as follows: *define, clear, set, load,* and *store.* If the value in a specified predicate register is true, the instruction is executed normally; if the value is false, the instruction is suppressed.

Predicate register values may be set using predicate define instructions. The predicate define semantics used are those of the HPL Playdoh architecture [20]. There is a predicate define instruction for each comparison opcode in the original instruction set. The instruction format of a predicate define in the IMPACT intermediate representation is shown below.

$$\texttt{pred\_}<\textit{cmp}>\ <\mathrm{P}_{in}>\ \texttt{[(}\mathrm{P}_{out1}\ <\textit{type}>\texttt{)}\ \texttt{(}\mathrm{P}_{out2}\ <\textit{type}>\texttt{)][(src1)}\ \texttt{(src2)]}$$

This instruction assigns values to registers P_{out1} and P_{out2} according to a comparison of *src1* and *src2* specified by <*cmp*>. The comparison <*cmp*> can be equal (eq), not equal (ne), greater than (gt), etc. A predicate <*type*> is specified for each destination predicate. Predicate defining instructions can also be predicated, as specified by P_{in}. An example of a predicate defining instruction is shown in Figure 5.14(a). This instruction is emulated with the sequence of C statements as shown in Figure 5.14(b).

```
(op 242 pred_ge <(r 102 p)> [(r 105 p_ot)(r 103 p_uf)]
                             [(r 4 i)(i 127)])
                           (a)

      if ( PredR[102] ) {
            PredR[105]  |= (IR[4] >= 127);
            PredR[103]  = !(IR[4] >= 127);
       }
       else{
           PredR[103]  = 0;
       }
                           (b)
```

Figure 5.14: A predicate define instruction in: (a) *Lcode* form, and (b) C emulation form

The predicate <*type*> determines the value written to the destination predicate register based upon the result of the comparison and of the input predicate, P_{in}. The following are predicate types which are particularly useful: unconditional (U), conditional, OR, and AND type predicates and their complements. The conditional predicate type is not used in the IMPACT compiler. Table 5.1 contains the truth table for these predicate types, with the exception of the conditional type.

Table 5.1: Predicate definition truth table.

		P_{out}					
P_{in}	Comparison	U	$\overline{U}$	OR	$\overline{OR}$	AND	$\overline{AND}$
0	0	0	0	-	-	-	-
0	1	0	0	-	-	-	-
1	0	0	1	-	1	0	-
1	1	1	0	1	-	-	0

Unconditional destination predicate registers are always defined, regardless of the value of P_{in} and the result of the comparison. If the value of P_{in} is 1, the result of the comparison is placed in the predicate register (or its complement for $\overline{U}$). Otherwise, a 0 is written to the predicate register. OR-type destination predicate registers are set if P_{in} is 1 and the result of the comparison is 1 (0 for $\overline{OR}$), otherwise the destination predicate register is unchanged. AND-type predicates are analogous to the OR-type predicate. AND-type destination predicate registers are cleared if P_{in} is 1 and the result of the comparison is 0 (1 for $\overline{AND}$), otherwise the destination predicate register is unchanged.

While predication is a part of the IMPACT instruction set architecture, it is not supported on modern processors. Therefore, the IMPACT compiler emulates predicated *Lcode* instruction, inserting control flow into the program. This is the same approach used by the *Lcode* emulator, since there is no means to specify that an instruction be predicated in a high-level language. An example of a predicated instruction in the *Lcode* format and the resulting C statements are shown in Figure 5.15

```
(op 53 ld_i <LF> <(r 104 p)> [(r 27 i)] [(r 72 i)(i 0)]
            <(label (l _wordct)( i 0))(wgt (f 0.156))>)
                         (a)

            if ( PredR[104] ) {
                 temp_var = IR[72];
                 IR[27]  = *((int *) temp_var);
            }
                         (b)
```

Figure 5.15: A predicated instruction in: (a) *Lcode* form, and (b) C emulation form

5.3.2 Speculation

Speculative execution is another architectural feature that is used to overcome the limitations imposed by branches in a program. Speculative execution involves the promotion of an instruction above the guarding branch. An instruction that should not be reached in the normal control flow of the program can be referred to as a potentially excepting instruction. The general speculation model [26] is used to handle speculated instructions that raise exceptions during the execution of the program. In this model, the architecture provides a silent version of instructions that may potentially cause an exception. Thus, exceptions resulting from speculated instructions are ignored; i.e., only exceptions from nonspeculative instructions are handled or could cause program termination. In order to emulate speculative instructions in a high-level language, an exception handler that would ignore exceptions in certain cases is required.

Emulation of speculative instructions is one of the areas where the *Lcode* emulator departs from the notion of architectural independence. A signal handler is used to provide the facilities for handling exception conditions that arise during execution. Generally, when the handler returns, execution would resume at the instruction that raised the exception signal. Hence, the requirement of ignoring exceptions requires moving the instruction pointer to the next instruction in the program. This exception-handling capability is provided by an architecture-specific module. The module is linked into the C source code to aid in the emulation of general speculation

For a program that contains speculative instructions, the *Lcode* emulator includes the header file *signal.h* in the C source code that is generated. This header file contains definitions of the C run-time library functions used for handling various conditions that arise during program execution. The library function used in the emulation of speculative instructions is *signal()*. This function determines how subsequent expception signals will be handled. It takes two arguments, signal number, *sig*, and a function pointer, *handler*. When the signal *sig* occurs, the signal is restored to its default behavior. Hence, the handler function is un-installed. Then the function pointed to by *handler* is invoked. If the *handler* function returns, execution will resume where it was when the signal occurred. Based on this knowledge of the exception handling capabilities of C, the *Lcode* emulator installs a signal handler for handling speculative load instructions as follows:

```
signal(SIGSEGV, signal_handler);
```

This handler is used to ignore segmentation faults that might result from illegal storage access. When this exception subsequently occurs, the signal handler un-installs itself. Thus, the signal handler module has the ability to re-install itself after handling the exception condition. The initial installation of the signal handler is performed upon entry into the *main* function of the program.

The flags attached to an *Lcode* instruction determines its speculation status. Without digressing into details about the flags used in the IMPACT *Lcode*, several different flag attributes are available for use on speculated instructions. The flag attributes of interest are "safe potentially excepting instruction," F, and "mask potentially excepting instruction," M. Instruction marked with the former flag are safe based on analysis in [16]; therefore, suppression is not required. Instructions marked with the latter flag are the only instuctions that might potentially raise an exception, hence requiring special exception handling. As shown in Figure 5.16(a), the instruction has the mask potentially excepting instruction flag, M, set. Emulating *Lcode* with speculative instructions involves setting a global mask variable before the speculated instruction. The mask is cleared immediately following the speculated instruction. This enables the exception handler to determine which exceptions should be handled or ignored. An example of this implementation is shown in Figure 5.16(b).

```
(op 34 ld_i <M> [(r 8 i)] [(r 33 i)(i 4)])
                   (a)
```

```
MASK_EXCEPTION = 1;
temp_var = IR[33]  + 4;
IR[8]  = *((int *) temp_var);
MASK_EXCEPTION = 0;
          (b)
```

Figure 5.16: Speculative load in: (a) *Lcode* form, and (b) C emulation form

The signal handler approach is only used for handling exception resulting from speculative load or store instructions. The only other speculative instruction that could potentially raise an exception is a divide instruction. This could result from using an invalid divisor of zero. The *Lcode* emulator handles this instruction by inserting an if-statement that tests the divisor for zero before the actual execution of the speculated divide instructions, as shown in Figure 5.17(b). With respect to speculation support, this is the only opportunity to provide an architecture-independent solution in the emulation of the IMPACT *Lcode*.

```
(op 37 div <M> [(r 3 i)] [(r 8 i)(r 4 i)])
                        (a)

if( IR[4] != 0 )
    IR[3] = IR[8] / IR[4];
                        (b)
```

Figure 5.17: Speculative divide in: (a) *Lcode* form, and (b) C emulation form

5.4 Overview

This chapter has provided a detailed description of the process involved in emulating the IMPACT *Lcode* in C. In addition, detailed descriptions of the mechanisms used to handle special architectural features has been presented. In order to put all this effort into perspective, this section presents the overall structure of the emulation process using the UNIX utility word-count, *wc*. The *wc* program consist of global data variables and three functions: *main*, *wcp*, and *ipr*. The *Lcode* versions of these functions can be split into one function per source file, or they can all reside in the same source file. The only requirement regarding the placement of *Lcode* in files is that the data section for global data must reside in a file by itself. Figure 5.18 shows *wc* split into three function files and one data file.

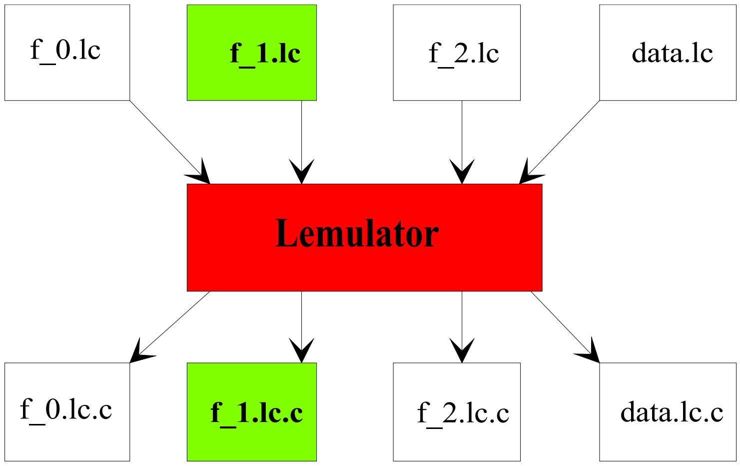

Figure 5.18: Breakdown of the *wc* program

With these files as input, the *Lcode* emulator produces corressponding C files that can be compiled with any generic C compiler. The *Lcode* function *wcp* is shown in Figure 5.19, and its equivalent C source code is shown in Figure 5.20.

```
(ms text)
(global _wcp)
(function _wcp 0.950000 <E> <
(ARCH:HPPA)
(MODEL:PA_7100)>)
  (cb 1 0.950000 [(flow 0 2 0.950000)] <(trace (i 2))>)
    (op 1 define [(mac $P0 i)] [])
    (op 2 define [(mac $P1 i)] [])
    (op 3 define [(mac $P2 i)] [])
    (op 4 define [(mac $P3 i)] [])
    (op 5 define [(mac $return_type i)] [])
    (op 6 define [(mac $local i)] [(i 0)])
    (op 7 define [(mac $param i)] [(i 16)])
    (op 8 prologue [] [])
    (op 9 mov [(r 1 i)] [(mac $P0 i)])
    (op 10 mov [(r 2 i)] [(mac $P1 i)])
    (op 11 mov [(r 3 i)] [(mac $P2 i)])
    (op 12 mov [(r 4 i)] [(mac $P3 i)])
  (cb 2 0.950000 [(flow 1 9 0.000000)(flow 0 3 0.950000)] <(trace (i 2))>)
    (op 13 ld_c [(r 5 i)] [(mac $P0 i)(i 0)] <(param (i 0))>)
    (op 14 beq [] [(r 5 i)(i 0)(cb 9)] <(NL_stln)>)
  (cb 3 2.850000 [(flow 108 4 0.950000)(flow 0 12 1.900000)] <(trace (i 1))>)
    (op 17 ld_c [(r 7 i)] [(r 1 i)(i 0)] <(param (i 0))>)
    (op 47 add [(r 1 i)] [(r 1 i)(i 1)])
    (op 18 beq [] [(r 7 i)(i 108)(cb 4)] <(NL_inner)>)
  (cb 12 1.900000 [(flow 119 8 0.950000)(flow 0 13 0.950000)] <(trace (i 1))>)
    (op 19 beq [] [(r 7 i)(i 119)(cb 8)] <(NL_inner)>)
  (cb 13 0.950000 [(flow 1 5 0.000000)(flow 0 7 0.950000)] <(trace (i 1))>)
    (op 20 bne [] [(r 7 i)(i 99)(cb 5)] <(NL_inner)>)
  (cb 7 0.950000 [(flow 1 5 0.950000)] <(trace (i 1))>)
    (op 29 mov [(mac $P0 i)] [(r 2 i)])
    (op 30 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
  (cb 5 2.850000 [(flow 1 3 1.900000)(flow 0 9 0.950000)] <(trace (i 1))>)
    (op 25 ld_c [(r 9 i)] [(r 1 i)(i 0)] <(param (i 0))>)
    (op 26 bne [] [(r 9 i)(i 0)(cb 3)] <(LB_inner)(LE_inner)>)
  (cb 9 0.950000 [] <(trace (i 0))>)
    (op 27 epilogue [] [])
    (op 28 rts [] [] <(tr (mac $P15 i))>)
  (cb 4 0.950000 [(flow 1 5 0.950000)] <(trace (i 3))>)
    (op 22 mov [(mac $P0 i)] [(r 4 i)])
    (op 23 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 46 jump [] [(cb 5)] <(NL_inner)>)
  (cb 8 0.950000 [(flow 1 5 0.950000)] <(trace (i 4))>)
    (op 33 mov [(mac $P0 i)] [(r 3 i)])
    (op 34 jsr <E> [] [(l _$fn_ipr)] <(tr (mac $P0 i))(ret (mac $P15 i))(param_size (i 16))>)
    (op 36 jump [] [(cb 5)] <(NL_inner)>)
(end _wcp)
```

Figure 5.19: *_wcp* function of *wc* in the *Lcode* format

```
#include "extern.h"
/* declare external variables */
extern int *global_OP_ptr;

int wcp(P0, P1, P2, P3)
int P0;
int P1;
int P2;
int P3;
{
	/* Local variable declarations */
	unsigned temp_var;
	int P15;
	int IP0;
	int IR[10];
	/* outgoing parameter space declaration */
	int OP[4];
	int *previous_OP_ptr;
	/*saving global OP pointer, and init global_OP*/
	previous_OP_ptr = global_OP_ptr;
	global_OP_ptr = (int *)((unsigned) OP + 12);
CB_1:
	IR[1]  = P0;
	IR[2]  = P1;
	IR[3]  = P2;
	IR[4]  = P3;
CB_2:
	temp_var = P0;
	IR[5]  = *((char *) temp_var);
	if (IR[5]  == 0) goto CB_9;
CB_3:
	temp_var = IR[1] ;
	IR[7]  = *((char *) temp_var);
	IR[1]  = IR[1]  + 1;
	if (IR[7]  == 108) goto CB_4;
CB_12:
	if (IR[7]  == 119) goto CB_8;
CB_13:
	if (IR[7]  != 99) goto CB_5;
CB_7:
	P0 = IR[2] ;
	P15 = ipr(P0);
CB_5:
	temp_var = IR[1] ;
	IR[9]  = *((char *) temp_var);
	if (IR[9]  != 0) goto CB_3;
CB_9:
	global_OP_ptr = previous_OP_ptr;
	return P15;
CB_4:
	P0 = IR[4] ;
	P15 = ipr(P0);
	goto CB_5;
CB_8:
	P0 = IR[3] ;
	P15 = ipr(P0);
	goto CB_5;
}
```

Figure 5.20: C code produced for the *_wcp Lcode* function of *wc*

6. THE *LCODE* EMULATOR TOOL

The internal structure and the implementation of the *Lcode* emulator has been described in the detail in the preceeding chapters. This chapter presents some of the other capabilities provided by the *Lcode* emulator tool. In addition, it also provides a guide to using the tool.

6.1 Integration with Existing IMPACT Tools

As described in section 3.1, the IMPACT simulator employs a trace-driven mechanism for simulating system or hardware models. This mechanism allows for profiling and simulation. An alternative implementation to the profiling capabilities of the IMPACT simulator has been presented in this thesis. However, the *Lcode* emulator's functionality does not include cycle-by-cycle simulation in a stand alone manner. This functionality is not included because simulation is not critical for the generation of highly optimized code. On the other hand, it is useful for evaluating performance of the code generated and its effect on the hardware. Hence, the *Lcode* emulator has the capability to generate

C source code that interacts with the IMPACT simulator. The simulator creates a UNIX pipe between itself and the program under simulation. The pipe is a communication path, consisting of a FIFO queue of bytes, between two processes.

In order to generate C source code for simulation, the *Lcode* emulator simply inserts a function call to send tokens through a pipe to the simulator. The tokens are integer values used to distinguish the program characteristics that are of interest to the simulator. The simulator is used to trace the following in a program: control flow, load and store addresses, hashing jumps, predicate values, predicate defining instructions, and predicated jumps. Communication through the pipe can be time consuming if the frequency of communication is high. The *Lcode* emulator assumes that the frequency of function calls that send tokens through the pipe to be high. In order to alleviate this problem, the emulated code stores the tokens in a token buffer. After a specified treshhold is reached, the token buffer's content is sent through the pipe for the simulator to process. Since the simulator can be used to trace the control flow of a program, the *Lcode* emulator can also use it to generate profile information. In fact, probe insertion for simulation is merely an extension of probe insertion required for profiling.

6.2 Using the *Lcode* Emulator Tool

There are two ways to run the *Lcode* emulator tool. The first is to directly run *Lemulator*, which is the name of the *Lcode* emulator tool executable. The second is to run the *gen_Lemulator* script, which calls *Lemulator* and sets the necessary parameters.

The latter approach is generally preferred, as it is easier to use, and requires less command line arguments. Both methods of running the tool are described in the following sections.

6.2.1 *Lemulator*

The command line syntax of a call to Lemulator is as follows:

```
Lemulator [-i infile] [-o outfile] [-p parmfile] [-Pmacro=value]
          [-Fparm=value]
```

The list of *Lcode* files to process is specified with *infile*. This list also serves as the foundation for the C source code generated. Each *Lcode* file name is concatinated with '.c' to form its corresponding C source file name. The output file is specified with *outfile*, which is only required for probing for simulation or profiling. The output file is an address list consist of function names in the program and a unique identifier assigned by the *Lcode* emulator. The parameter file to be used is specified by the *parmfile*. If it is not specified, the parameter file specified by the STD_PARMS_FILE environment variable is used. If the environment variable is not set, the STD_PARMS file in the current directory is used.

The option '-Pmacro=value' is used to define a *macro* as *value* for parameter file preprocessing. The *macro* may used to define parameter values. A *macro* specification is not required for running the *Lcode* emulator. All optional parameters to the tool are specified using the option '-Fparm=value'. This option is used to force *parm* to *value* in the parameter file. The specific parameters for the *Lcode* emulator are as follows:

DEBUG: This switch turns on a verbose printing of status information during the generation of C source code. The debug information is printed to a file, and the default file name is *out.debug*. In addition, each line of C source code generated is tagged with a commented number corresponding to the *op* number of its equivalent *Lcode* instruction.

Speculated_code: Specifies the presence of speculative instructions in the *Lcode* files.

Stand_alone_profiling: Specifies the type of profiling function calls to insert in the body of the C source code generated.

Profiling: Similar to the *Stand_alone_profiling* option.

Simulation: Similar to the *Stand_alone_profiling* option.

encoded_file: Name of the encoded file used with *Stand_alone_profiling*. The encoded file is an image of the program, and the default file name is *out.encoded*.

Output_Lcode_Ccode_mapping: Generates a table with a mapping of lines in the *Lcode* file to lines in the output C source code files. This table is stored in a file named *table.map*.

6.2.2 *gen_Lemulator*

Obviously, specifying all of the necessary parameters with the correct syntax could become tedious. Therefore, the *gen_Lemulator* script was written to simplify running the *Lcode* emulator tool. The command line syntax used to call *gen_Lemulator* is:

```
gen_Lemulator "dir" "benchmark" "list" [options]
```

for which the arguments are as follows:

dir: Specifies the directory where the input files are located.

benchmark: Specifies the name of the benchmark being processed.

list: File containing a list of all the *Lcode* files to be processed.

options: The options that can be specified are as follows:

- **debug:** same as *DEBUG* in section 6.2.1
- **mapping:** same as *Output_Lcode_Ccode_mapping* in section 6.2.1
- **speculation:** same as *Speculated_code* in section 6.2.1
- **fast_prof:** same as *Stand_alone_profiling* in section 6.2.1
- **profiling:** same as *Profiling* in section 6.2.1
- **simulation:** same as *Simulation* in section 6.2.1
- **merge:** Merge profile information into orignal *Lcode* files

If the *Lcode* files are being processed for simulation or profiling, this script invokes another tool, *Lencode*, to create an encoded image of the *Lcode* files. *Lemulator* is called once to process all the files specified in *list*, with the given options. The *outfile* and *parmfile* arguments to *Lemulator* are not specified, therefore default values are used. After the *Lemulate* generates the C source files, this script invokes a generic C compiler to build an executable from the C source files. If the executable has profiling or simulation capabilities embedded, the script would run the executable with the appropriate arguments. The output of the program is checked against the expected output to show correctness. Finally, if the *merge* option is specified, the profile information generated is merged into the original *Lcode* files by invoking the *gen_Lget* script.

7. CONCLUSION

This thesis has presented an emulation environment as an alternative to some aspects of the IMPACT simulator. The emulation process involves translating the low-level IMPACT intermediate representation into equivalent high-level language statements. The high-level language used in this project is the C programming language. Emulating the IMPACT intermediate representation in C provides a high level of portability to the compilation process.

As profiling is an important aspect of generating efficient code in the IMPACT compiler, this emulation environment provides a platform-independent means of profiling IMPACT *Lcode*. This profiling capability is provided by profile counters that augment the body of the profiled program. This stand-alone approach for profiling is used solely to support control flow profiling. The platform independence is provided as a result of being able to use a generic C compiler to produce an executable from the C source code generated by the *Lcode* emulator. The complexity involved in generating C source code for profiling or simulation is minimal compared to the traditional IMPACT implementation. In order to establish a link between the *Lcode* emulator and the suite of available IMPACT tools, the capability to interact with the IMPACT simulator is provided.

Nontraditional architectural features, such as predication and speculation, which only exist in research architectures, are supported by the *Lcode* emulator. In addition, emulation of *Lcode* instruction in C suppliments the IMPACT compiler with a rich set of C source-level debuggers. The *Lcode* emulator presented in this thesis has been sucessfully tested on SPEC integer benchmarks and several UNIX utilities. In comparison to the HP-PA specific implementation of profiling, the stand-alone capability of the *Lcode* emulator runs about 38% faster. This tool should allow for extensions to the current IMPACT ISA and verification of the intermediate representation without the high implementation time required for a code generator.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[2] W. Y. Chen, S. A. Mahlke, N. J. Warter, R. E. Hank, R. A. Bringmann, S. Anik, D. M. Lavery, J. C. Gyllenhaal, T. Kiyohara, and W. W. Hwu, "Using profile information to assist advanced compiler optimization and scheduling," in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[3] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of the 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[4] A. Krall, "Improving the semi-static branch prediction by code replication," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementaton*, pp. 97–106, June 1994.

[5] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, October 1994.

[6] P. A. Darnell and P. E. Margolis, *C, A Software Engineering Approach.* New York: Springer-Verlag, 1990.

[7] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, University of Illinois, Urbana, IL, 1992.

[8] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, University of Illinois, Urbana, IL, 1993.

[9] B.-C. Cheng, “Pinline: A profile-driven automatic inliner for the impact compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1997.

[10] D. M. Gallagher, “Memory disambiguation to facilitate instruction-level parallelism compilation,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[11] G. E. Haab, “Data dependence analysis for Fortran programs in the IMPACT compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1995.

[12] S. A. Mahlke, “Design and implementation of a portable global code optimizer,” M.S. thesis, University of Illinois, Urbana, IL, 1991.

[13] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, 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.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective compiler support for predicated execution using the hyperblock,” in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[15] R. E. Hank, “Machine independent register allocation for the IMPACT-I C compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1993.

[16] R. A. Bringmann, “Compiler-controlled cpeculation,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[17] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “The importance of prepass code scheduling for superscalar and superpipelined processors,” *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

[18] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, “Sentinel scheduling for superscalar and VLIW processors,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238–247, October 1992.

[19] N. J. Warter, “Modulo scheduling with isomorphic control transformations,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1993.

[20] V. Kathail, M. S. Schlansker, and B. R. Rau, “HPL PlayDoh architecture specification: Version 1.0,” Hewlett-Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-93-80, February 1994.

[21] J. C. Gyllenhaal, “A machine description language for compilation,” M.S. thesis, University of Illinois, Urbana, IL, 1994.

[22] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[23] 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, Tech. Rep. CRHC-91-1, January 1991.

[24] M. Loukides and A. Oram, *Programming with GNU Software.* Sebastopol, CA: O'Reilly, 1996.

[25] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 92–103, December 1997.

[26] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.