Technical Report

ExtractCFG: A Framework to Enable Accurate Timing Back Annotation of C Language Source Code

Arindam Goswami and Andreas Gerstlauer

UT-CERC-11-02

August 1, 2011

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

1 University Station, C8800
Austin, Texas 78712-0323
Telephone: 512-471-8000
Fax: 512-471-8967
http://www.cerc.utexas.edu

THE UNIVERSITY OF TEXAS AT AUSTIN
UT ECE
ELECTRICAL & COMPUTER ENGINEERING

ExtractCFG: A Framework to Enable Accurate Timing Back Annotation of C Language Source Code

Arindam Goswami, Andreas Gerstlauer
Computer Engineering Research Center
Electrical and Computer Engineering
The University of Texas at Austin

August 2011

The current trend in embedded systems design is to move the initial design and exploration phase to a higher level of abstraction, in order to tackle the rapidly increasing complexity of embedded systems. One approach of abstracting software development from the low level platform details is host-compiled simulation. Characteristics of the target platform are represented in a host-compiled simulation model by annotating the high level source code. Compiler optimizations make accurate annotation of the code a challenging task. In this thesis, we describe an approach to enable correct back-annotation of C code at the basic block level, while taking compiler optimizations into account.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, the rising complexity and the tight development schedules of embedded systems have thrown up new challenges in their hardware and software development. The generally accepted solution to tackle this complexity is to move system design and exploration to a higher level of abstraction [5]. Although some components of the design and final evaluation need to be done at low levels, there are large productivity gains in performing initial design space exploration and refinement at a higher level [7].

With ever increasing software content, the major component of cost in embedded system design is software development [16]. Within this process, designers rely heavily on simulations to validate and optimize software at early stages of the design process. One approach of abstracting software development from the low level platform details is host compiled simulation [6], [15]. In this approach software is simulated and tested on the host computer itself, and characteristics of the target platform are inserted into the host compiled simulation by annotating the high level source code. This approach is faster than simulation on traditional cycle-accurate processor models. Purely functional Instruction Set Simulators (ISS) can also provide faster simulation than cycle-accurate processor models. However, it is difficult to annotate ISS models with target platform characteristics while maintaining high simulation speeds.

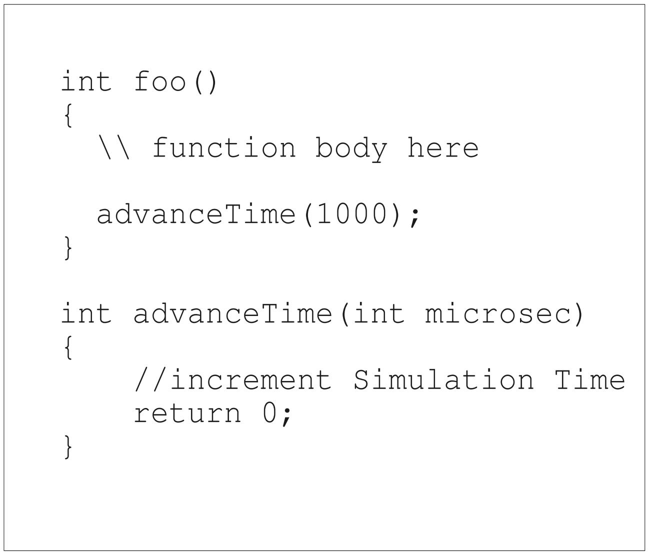
It is a common practice in host compiled simulation to instrument code with statements or functions that model simulation time (timing annotations). Such code annotations allow for the advantage of fast simulation, while retaining the concept of execution time on the target platform. This instrumentation approach may be used to annotate system power as well. However, accurate instrumentation of high-level source code with detailed, low-level target metrics is not a trivial task. This report deals with the design and implementation of a framework for the correct spatial placement of timing annotations in a C source file that describes the functionality of an embedded system.

The remainder of this chapter establishes the need for such a framework, and provides the requisite background knowledge on this subject.

1.1 Timing Annotation

Simulation languages like C, C++, SpecC and SystemC are commonly used to describe systems at a higher abstraction level. Though the functionality of a system can easily be tested or simulated in these languages, metrics like execution time are not inherently available at this level. The time the code takes to execute on the host machine running the simulation is not a very good indication of the actual execution time on the target platform. To bypass this, the original system definition code is instrumented at certain strategic points.

This is done by inserting a function that increments a variable that tracks system time. In dedicated System Level Design Languages (SLDLs) like SpecC and SystemC, there are keywords that implement this timing annotation very efficiently on their simulation engines. Figure 1.1 shows an example in which the function `advanceTime` is used to indicate that the function `foo` takes 1000 microseconds to execute. In an SLDL like SpecC or SystemC, the advanceTime function would be natively provided as a builtin `wait-for-time` keyword by the language and simulator.



```
int foo()
{
  \\ function body here

  advanceTime(1000);
}

int advanceTime(int microsec)
{
    //increment Simulation Time
    return 0;
}
```

Figure 1.1: An example of timing annotation.

The major challenges in inserting timing annotation into a high level design are estimation of the timing information, and deciding where to accurately insert the timing annotation.

1.1.1 Estimating Execution Time

Estimation of the execution time of high level code on a target platform, might be done by static analysis of the generated assembly, or by running the code on a cycle-accurate simulator of the target hardware. Although these estimation methods are time consuming, they need to be done just once for each code section, while the annotated code can then be used repeatedly. For long simulations, where the same piece of code executes multiple times, or for designs that require simulations to run repeatedly with different inputs, this approach can still save significant development time.

1.1.2 Granularity of Timing Annotations

The focus of this report is on the locations in the original high level code, where timing annotations may be appropriately inserted. We may choose to annotate timing information at different granularity levels. Generally, if the frequency of these annotations in the code is high, the simulation is slower, but the accuracy is higher. We now discuss the different levels of granularity at which code can be annotated.

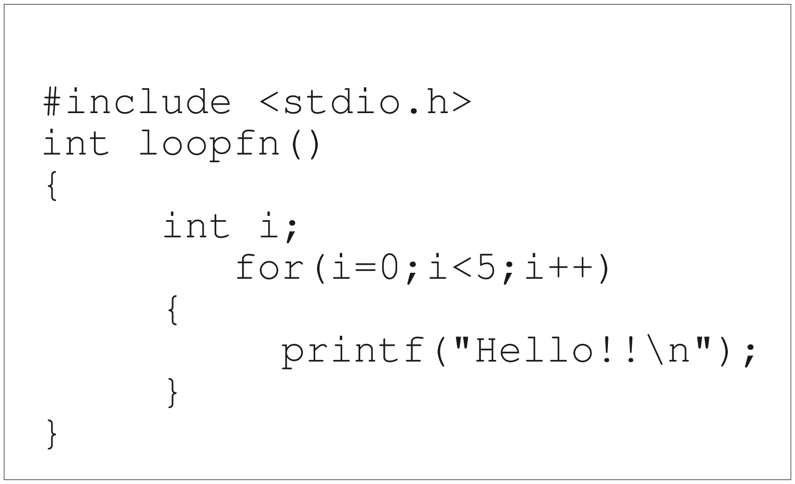
Timing may be annotated at the function level [3],[20]. Here, each function has one timing annotation. This approach slows down simulation speed the least. However, if a function has conditional statements or conditional loops, then the execution time of the function can vary by a large margin each time the inputs to the function or the execution environment change. Thus this approach is not very accurate.

At a finer granularity, timing annoations can be inserted at the basic block level. A basic block of code is defined as a segment of code that always executes in sequence and has just one entry and exit point [2]. In other words, it is a piece of code that has no branch statements in it. Timing annotation at the basic block level [17],[21] is more accurate than that at the function level. Variable execution time due to conditional statements can be taken into account. This is the level of timing annotation that the ExtractCFG framework is designed for. A Control Flow Graph (CFG) is the graph obtained by having basic blocks as nodes and the transition of control from one basic block to another as the edges. This is the reason for naming the framework ExtractCFG.

Timing annotation can also be done after each individual statement in the code. Though this is a more accurate approach, it takes much more time to simulate while not providing significantly improved accuracy compared to a block-level annotation.

1.2 Timing Annotation Challenges

Annotating timing at the basic block level is not a simple task. The C language syntax can make it inconvenient to accurately insert annotation statements into the source code. There is also the issue of compiler optimizations. Compilers can transform the CFG of the source code during the optimization process. Code within the basic block itself is also transformed. We now discuss these two challenges to timing annotation in more detail.



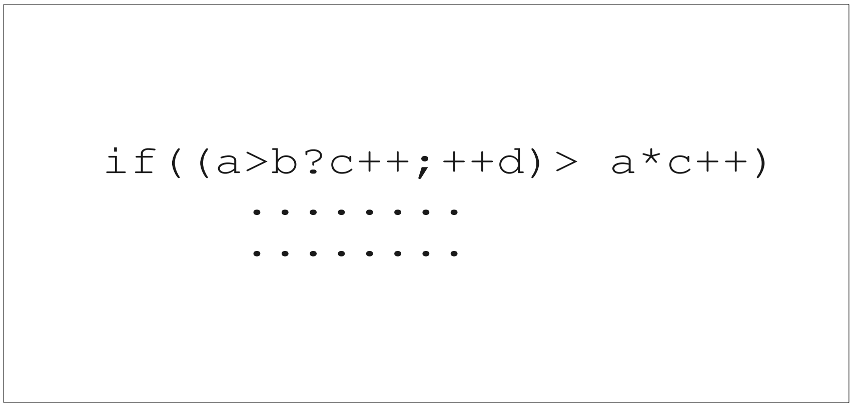
```
#include <stdio.h>
int loopfn()
{
     int i;
        for(i=0;i<5;i++)
     {
          printf("Hello!!\n");
     }
}
```

Figure 1.2: A `for` loop example.

1.2.1 Syntactical Problems

Constructs like `while` and `for` make it difficult to locate the basic blocks in the code. An example of a `for` loop is shown in Figure 1.2. In this loop, the initialization, the main body of the `for` loop and the conditional check in the `for` loop are three different basic blocks. In the current code, it is inconvenient to annotate these three basic blocks. Similarly, it is inconvenient to annotate code with `while` or `do while` loop structures.

In Figure 1.3, another example of a complex if statement is shown. In this example, the `if` statement has multiple basic blocks, in its condition field. Annotating this `if` statement is complicated in the current representation. There are actually four basic blocks in the condition itself. This is not counting the basic blocks introduced by the `if` statement.



```
if((a>b?c++;++d)> a*c++)
     .........
     .........
```

Figure 1.3: A complicated if statement.

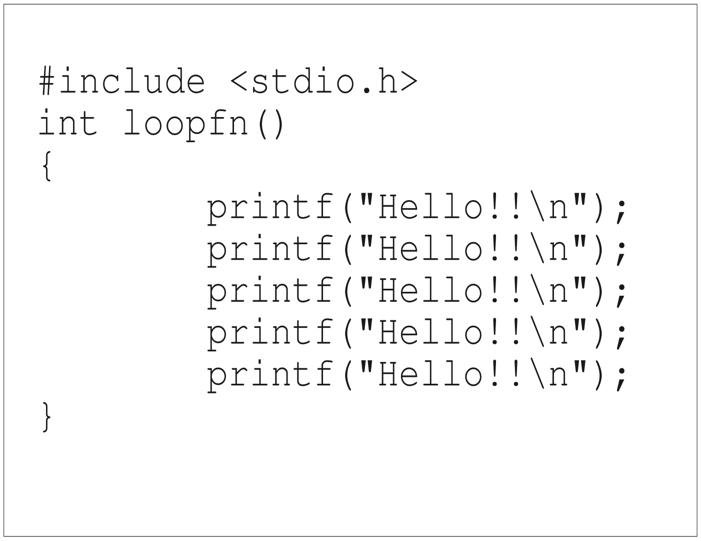
Instead of annotating code in the source representation, we can let the compiler expose the basic blocks for us. No modification of the compiler is required for this. This makes back annotating timing into the code significantly easier and allows for simpler automation of the task of timing annotation. The use of the compiler for this task is discussed in a later section.

1.2.2 Problems Due to Compiler Optimization

Compilers are usually set up to perform certain optimizations on the code. The compiler can be set up to optimize time, code size in memory, or both execution time and code size. These optimizations change the structure of the code, while maintaining its functionality. This might cause the actual basic blocks in the object file or executable to be different than the ones in the original source code.

Compiler optimization makes annotating time in the original source code a difficult activity, because the code structure that we see in the original source file does not map to the actual code structure in the optimized target code. As such, there is no clear relationship between estimates of execution time obtained for blocks in the target binary and appropriate locations for back-annotating such estimates into the source code.

Optimizations like loop unrolling, dead code elimination and function inlining can change the structure of the code in this way. The for loop example in Figure 1.2 could be optimized by loop unrolling to the form shown in Figure 1.4. In Figure 1.4, the loop was unrolled five times. The CFG of the original source with three basic blocks was reduced to a CFG with just one basic block.



```
#include <stdio.h>
int loopfn()
{
        printf("Hello!!\n");
        printf("Hello!!\n");
        printf("Hello!!\n");
        printf("Hello!!\n");
        printf("Hello!!\n");
}
```

Figure 1.4: An unrolled `for` statement.

1.3 Objective of the ExtractCFG Framework

As discussed in the previous section, C source code is difficult to annotate at the basic block level, due to the syntactical structure of the C code itself. Moreover, due to compiler optimizations, the basic blocks of the C code do not match the basic blocks of the machine code that is generated by the compiler. Hence, timing annotations in the original source code cannot accurately simulate the behavior of the code on the target processor. A version of the original code is needed with the same basic block structure or CFG of the compiler optimized code. We also need this representation to be in standard C code form, in order to enable host compiled simulation of final back-annotated code. For meaningful simulation, it is imperative that the converted C code has the same functionality as that of the original.

This is achieved by using intermediate source code that is dumped from the compiler front end after optimizations. The compiler here is a cross compiler that targets the final platform. The intermediate source code is then modified to get a compilable version of the original source code, which has post optimization basic blocks boundaries exposed. Due to a property of the compilation process, which will be discussed in the next chapter, the difficulty of annotating `for` loops, `while` loops, `do while` loops and complex conditional statements is also resolved.

Figure 1.5 shows how ExtractCFG fits into the host compiled simulation work flow. First, the original C code is processed by the compiler front end. The intermediate source code is dumped from the compiler after optimization

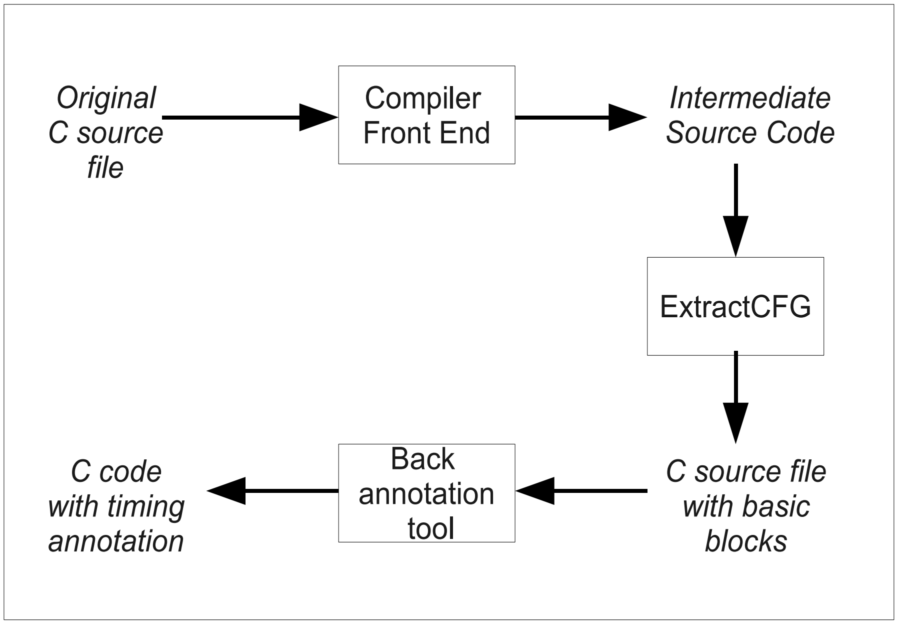


Figure 1.5: Overview of work flow.

has taken place. The intermediate representation now has a CFG matching that of the final machine code on the target processor. ExtractCFG processes this intermediate code and converts it into compilable C while maintaining the CFG structure. This C code has the same functionality as the original source, and the same basic blocks of the optimized code on the target machine. Furthermore, basic blocks are clearly exposed enabling accurate and possibly automated timing annotation.

The C code from ExtractCFG can now be back-annotated with timing information at the basic block level. The C code with back annotated timing information gives us a relatively accurate model for host compiled simulation.

The choice or design of the back-annotation tool is outside the scope of this report.

1.4 Design Decisions

This section briefly discusses some of the different design options for implementing the ExtractCFG framework, and the reasons for selecting a particular option.

1.4.1 Compiler Selection

GCC or The GNU Compiler Collection was originally designed as the C compiler for the GNU tool-chain. It has since grown to support a large number of programming languages like C, C++, Java, Fortran and ADA, and also provides standard libraries for these languages. It is a robust cross compiler that can target most common processor families like the ARM , Intel x86, PowerPC, BlackFin, MIPS, SPARC, VAX and Motorola 6800. GCC is widely used in embedded systems development, and for this reason we choose GCC as the supported compiler for the ExtractCFG Framework.

1.4.2 Scripting Language Selection

Since this project requires a lot of handling of text and extensive pattern matching, a 'scripting language' would be appropriate for its implementation. The languages Python, Perl and Ruby were considered. Ruby is not used because it is a comparatively new language, and lacks the level of documentation and community support that Python and Perl have. Both Python and Perl are powerful programming languages with extensive regular expression support. Finally Python was chosen because it has an arguably simpler syntax, and is more manageable for larger programs.

1.5 Related Work

There is significant existing research on obtaining the final CFG of source code. In [4], Ermedahl and Gustafsson use *abstract execution* to identify loop bounds and false paths in C code. This approach does not take into account all compiler optimizations. In [14], Puschner defines an approach to transform high level control flow information to the object code level. Aggressively optimized code cannot be handled by this approach .

In the domain of timing back-annotation for host-compiled simulation models, there are several, nearly identical approaches that work at the level of intermediate compiler representations. In [10], Hwang, Abdi and Gajski discuss the use of a datapath model of the target to back annotate timing information into intermediate source code from a LLVM compiler front end. Wang and Herkersdorf [21],[22] discuss the use of intermediate source code from GCC for back annotating timing information. However, implementation details for the conversion of intermediate source code into compilable C code are not provided, and it is difficult to judge the quality and accuracy of their approach.

1.6 Report Outline

The rest of the report is organized as follows: in Chapter 2, the GCC architecture is discussed and the intermediate representations of code at different compilation stages are analyzed. Chapter 3 deals with the design and implementation details of the ExtractCFG framework. The process of conversion of the intermediate representation back to compilable C code is also discussed in detail. In Chapter 4, we look at the test and validation procedures followed for the ExtractCFG project, and discuss the results. Finally, in Chapter 5, we discuss the conclusions drawn from the project, and mention possible future work for ExtractCFG.

Chapter 2

Compiler Front End

The compiler front end used in the ExtractCFG framework is GCC. The framework uses GCC to obtain an intermediate representation of source code after it has been optimized. This chapter discusses the GCC architecture and the intermediate representations of the code that GCC maintains at different stages of the compilation process.

2.1 GCC Architecture

The GCC compiler is composed of three major parts: a language dependent Front End, a language and target independent Tree Optimizer or Mid End, and a target-dependent Back End. This architecture was designed to facilitate contributors to the GCC project to add or modify optimization algorithms, processor support and language support independently, without having to modify other parts of the GCC source code.

Figure 2.1 gives an overview of the major components of the GCC architecture. The GCC Front End processes the source code files of the supported languages and produces a language-independent tree representation of the original code in a format called GIMPLE [18]. The Tree Optimizer then

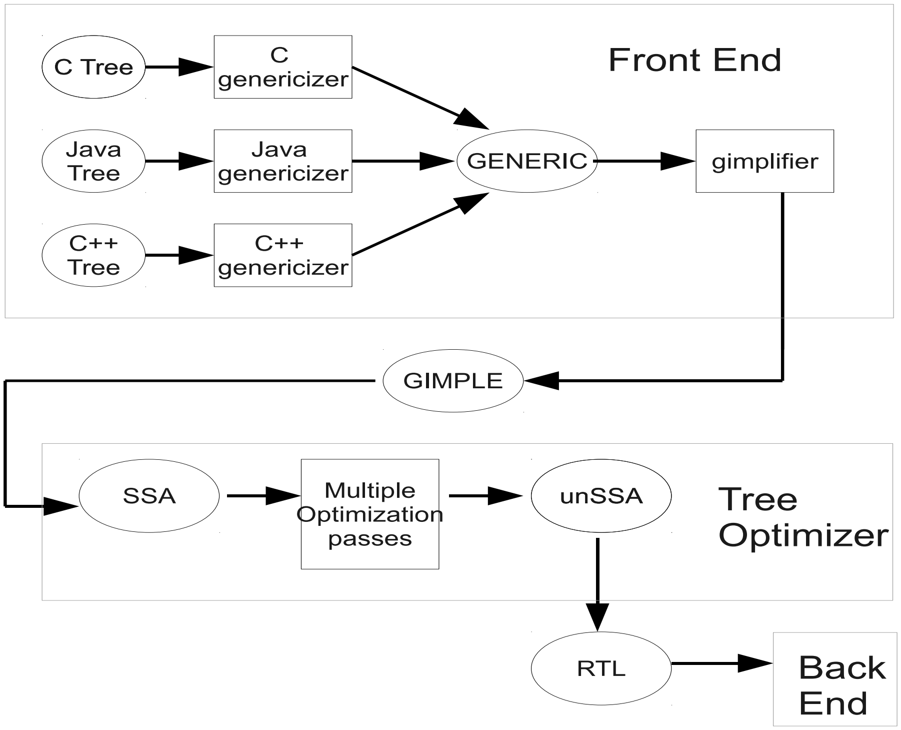
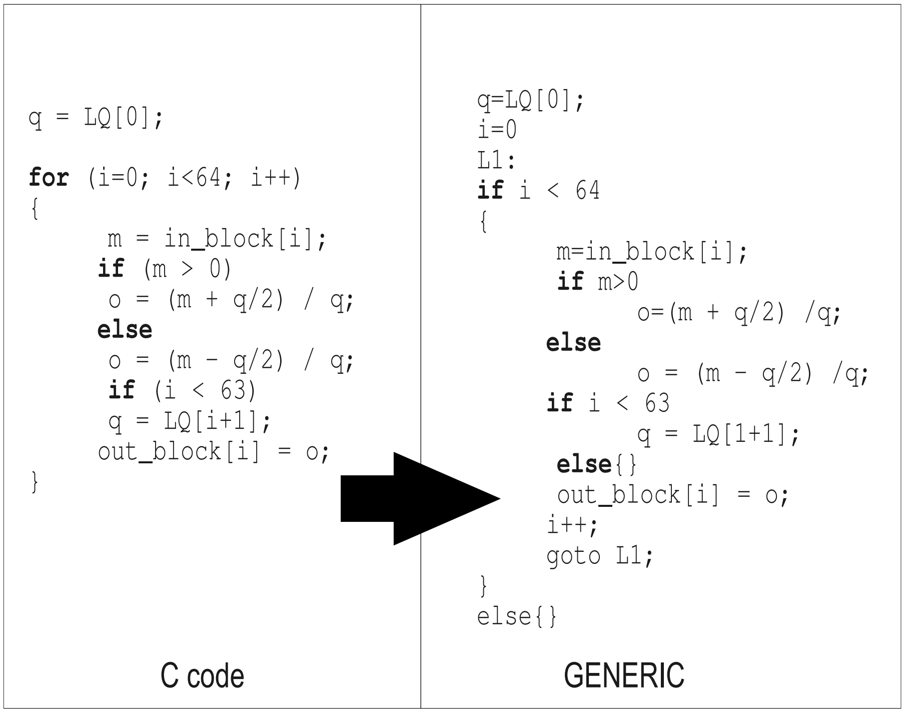


Figure 2.1: GCC compiler work flow [12].

performs hardware-independent optimizations on the GIMPLE tree and converts it into a Register Transfer Language (RTL) [18] representation. The Back End finally processes the RTL tree and generates machine code for the specific target. The Back End processing is target hardware dependent.

2.1.1 GCC Front End

The GCC front end is primarily designed to convert source code from different languages into a language-independent representation that enables



```
q = LQ[0];

for (i=0; i<64; i++)
{
       m = in_block[i];
      if (m > 0)
       o = (m + q/2) / q;
      else
       o = (m - q/2) / q;
       if (i < 63)
       q = LQ[i+1];
      out_block[i] = o;
}
```

```
q=LQ[0];
i=0
L1:
if i < 64
{
       m=in_block[i];
       if m>0
              o=(m + q/2) /q;
      else
              o = (m - q/2) /q;
      if i < 63
              q = LQ[1+1];
       else{}
       out_block[i] = o;
      i++;
      goto L1;
}
else{}
```

Figure 2.2: C and GENERIC tree comparison.

easier optimization of code. First, a parser converts input source code into a tree representation. The parser also checks for syntactical errors. This tree is not yet independent of the programming language of the source code, and each language has its own parser module. This language tree is then converted into a tree representation called GENERIC [18]. The GENERIC tree can represent all structures of any of the Front End programming languages, while itself being totally language-independent.

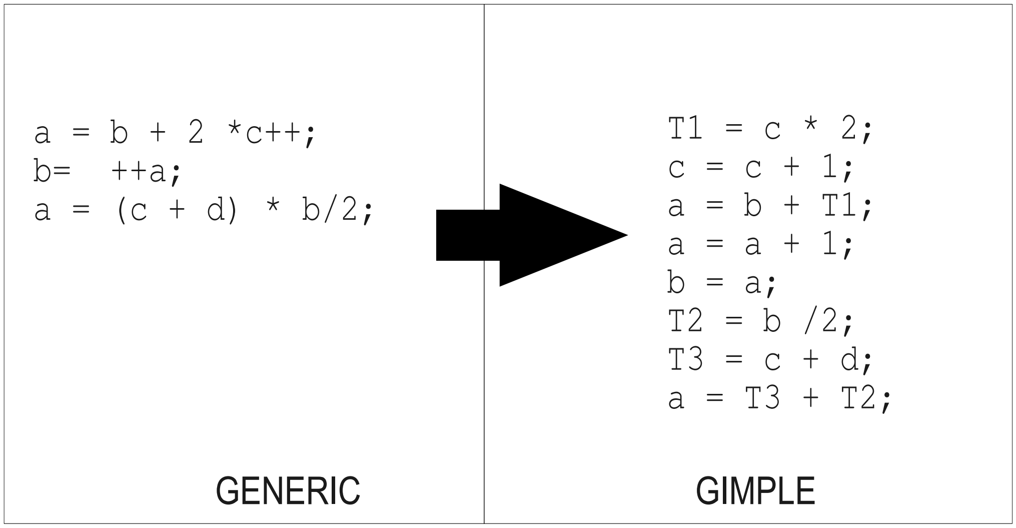
The example in Figure 2.2 compares C code with the GENERIC tree generated from it. We can see that the language dependent structures like `for` are not present in the GENERIC code. Instead we have `if-else` structures. This generic processing will later eliminate the problem of annotating structures like `for`, `while` and `do -while`.

2.1.2 GCC Tree Optimizer

The Tree Optimizer deals with three intermediate representations: They are 1) GIMPLE, 2) Control Flow Graph, and 3) SSA. We now discuss how the Tree Optimizer in GCC processes these intermediate representations and how it uses them to perform various optimization steps.

Although the GENERIC tree is language-independent, it is still too complex to enable easy optimization of the code. To handle this complexity, the GENERIC tree is converted into the GIMPLE format. GIMPLE is based on the SIMPLE representation, which was designed by the McCAT Project at McGill University [9]. It is a three-address representation, which is much simpler than the GENERIC format. GIMPLE allows for easy optimization in a language and target independent manner. The example in Figure 2.3 shows the GIMPLE format, and its simplicity as compared to the GENERIC representation. All statements in the GENERIC format are broken down into simple code with three operands each.

The GIMPLE tree is then converted to a Control Flow Graph (CFG) representation. The CFG has basic blocks as nodes and execution paths as edges. This representation is used by the GCC target-dependent Back End.



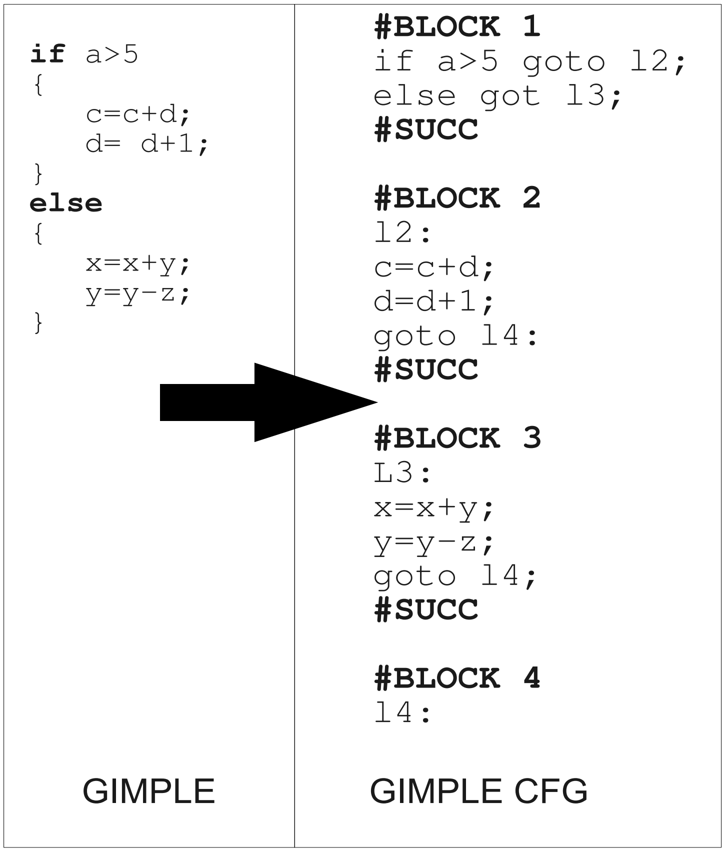
```
a = b + 2 *c++;
b=  ++a;
a = (c + d) * b/2;
```

```
T1 = c * 2;
c = c + 1;
a = b + T1;
a = a + 1;
b = a;
T2 = b /2;
T3 = c + d;
a = T3 + T2;
```

Figure 2.3: GIMPLE and GENERIC tree comparison.

The CFG is still in a GIMPLE format, but now it has the edges between basic blocks explicitly annotated. The GCC module that constructs the CFG also removes conditional statements like if(0) and if(1), and it eliminates nodes which are not traversed. The ExractCFG framework uses the basic blocks in the CFG to define possible locations for inserting timing annotations into the High Level C source code. Figure 2.4 shows an example of a CFG representation. The CFG representation has the basic block boundaries clearly marked with #BLOCK and #SUCC statements.

In the next step, the tree is changed to a Single Static Assignment (SSA) format [12]. Single Static Assignment [11], [12] is a three address representation, in which each variable is assigned a value only once. If a variable

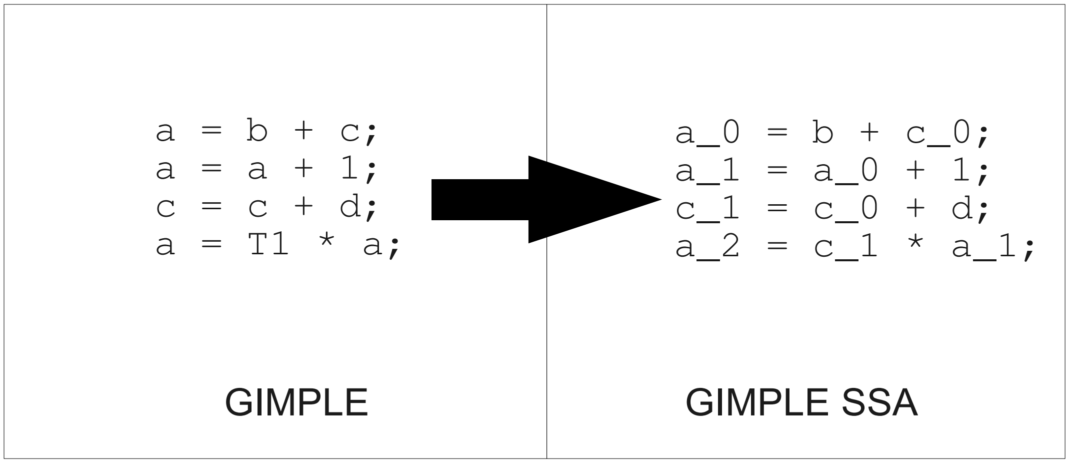


```
if a>5
{
   c=c+d;
   d= d+1;
}
else
{
   x=x+y;
   y=y-z;
}
```

```
#BLOCK 1
if a>5 goto l2;
else got l3;
#SUCC

#BLOCK 2
l2:
c=c+d;
d=d+1;
goto l4:
#SUCC

#BLOCK 3
L3:
x=x+y;
y=y-z;
goto l4;
#SUCC

#BLOCK 4
l4:
```

Figure 2.4: GIMPLE and CFG tree comparison.

has values assigned to it multiple times, a new copy of that variable is created at each assignment. References to a variable during the program flow refer to the copy of the variable that was last assigned to. An example of conversion of GIMPLE to the SSA representation is shown in figure 2.5. New copies are made of the variable a every time it is referenced. Only the newest copy of the variable is referred to in the code at any time.



```
a = b + c;
a = a + 1;
c = c + d;
a = T1 * a;
```

```
a_0 = b + c_0;
a_1 = a_0 + 1;
c_1 = c_0 + d;
a_2 = c_1 * a_1;
```

Figure 2.5: GIMPLE and SSA tree comparison.

Most optimizations in GCC take place on the GIMPLE SSA format, with separate passes for each optimization process. The SSA representation is still in GIMPLE format. The GCC Back End cannot handle an SSA tree, so the GCC SSA module converts the SSA back into a normal (nonSSA) representation after the optimizations take place. The ExtractCFG framework uses the tree after optimizations and after it has been converted back from SSA.

2.1.3 RTL Back End

The tree is finally converted into an RTL format [18]. This representation is very close to the the target hardware. In the Back End, the assembly code is generated. Although most optimizations take place on the GIMPLE tree, there are some optimizations that are performed at the RTL level. This may sometimes lead to minor changes between the GIMPLE Control Flow

Graph after optimizations, and the final RTL Control Flow Graph. However, the post-optimization GIMPLE CFG is generally identical to the final assembly.

Optimizations that take place in the RTL Back End are not taken into account by the ExtractCFG Framework. This is not a major issue as the majority of the optimizations are done in the Tree Optimizer, and the current trend of GCC development is to move more optimization passes into the Tree Optimizer stage [13]. In the next chapter, we look at an example of how to annotate code for an ARM target when the RTL back end has changed the structure of the CFG.

2.2 Intermediate Representations

In order to facilitate debugging, GCC enables the Internal Representations to be dumped to a file, at various stages in the compilation process. Certain options [19] can be used while invoking GCC in order to obtain such dumps. In this section, we look at some of these dump options, and their potential usefulness for the ExtractCFG Framework.

2.2.1 RTL Dump

The `-fdump-rtl-`*`pass`* option can be used with GCC to dump the RTL tree at various stages of the RTL processing. The *`pass`* in this option can be replaced with the names of the RTL pass or stage, we want the dump after. Some possible stages to dump the RTL tree are:

`-fdump-rtl-expand`: Dump after conversion to RTL.

`-fdump-rtl-dce`: Dump after RTL dead code elimination.

`-fdump-rtl-bbro`: Dump after RTL basic block reordering.

`-fdump-rtl-all`: Dump all possible RTL stages.

2.2.2 IPA Dump

The `-fdump-ipa-`*`switch`* dump option enables dumping during different stages of processing the Inter-Procedural Analysis (IPA) language tree in the Front End. The `switch` defines the stage of the IPA at which the tree can be dumped. IPA is a compiler optimization method, where the entire program is analyzed and not just one function.

2.2.3 GIMPLE Dump

The `-fdump-tree-`*`switch-options`* option can dump the intermediate tree at various stages. The word *`options`* specifies the level of detail we want in the dump. Some choices for the `options` fields are:

`raw`: Enables dumping a tree in a raw format. By default, trees are dumped using a 'pretty print' format, which resembles C. A 'pretty print' format is more readable by humans, and is somewhat closer to compilable C code.

`blocks`: Shows the basic block boundaries in the dump.

`all`: Dumps the tree in a 'pretty print' format, with additional information like basic blocks, and various pass statistics.

The word *`switch`* specifies the stage at which the dump is performed. Some choices are:

`gimple`: Dump the GIMPLE tree after it has been created.

`ssa`: Dump the GIMPLE SSA tree after creation.

`alias`: Dump all aliasing information. For example, if a pointer `‘int *ptr’` aliases an integer `‘int num’`, then `‘*ptr’` and `‘num’` are regarded as aliases, and this information is dumped in the alias file.

`optimized`: Dump the tree after all tree based optimizations.

`all`: Dump the tree at all possible stages.

2.2.4 Intermediate Representation Selection

GCC provides a number of choices for dumping the intermediate representation tree at different stages. We need to choose a representation that is close to the final source, and can still be converted back to C conveniently and fast.

The RTL (Register Transfer Language) tree representation reflects the final assembly code very well. However, its grammar is not at all similar to C. Even if RTL to C conversion were somehow implemented, it would be very time consuming.

The GIMPLE tree can be dumped by GCC in a ‘pretty print’ format, which is somewhat similar to C. Hence, a dump of the GIMPLE tree, after all optimizations have been performed, would suit our purpose. We use GCC with the `‘-fdump-tree-all-blocks-details’` option. This creates dump files at all stages of the GIMPLE tree optimization process. The dump files have the basic block structure exposed, and are in the ‘pretty print’ format.

We choose the dump file with the extension ‘.blocks’ , because it is dumped after all GIMPLE SSA optimizations. The ‘blocks’ intermediate representation file will hereafter be referred to as the IR file in the remainder of this report. To convert the IR file, a look-up table of the variables referenced by each function in the C file is required. This look-up table is extracted from the dump of the alias information in the dump file with extension ‘.alias’. Details of this process will be described in the next chapter.

Chapter 3

IR Conversion

The ExtractCFG Framework uses an Intermediate Representation (IR) of the source code, which is generated during compilation as described in the previous chapter. IR conversion translates the IR to compilable C code while preserving the control flow graph. This leaves us with a C source file with the same functionality as the original C file, but with a structure similar to that of the generated machine code. This chapter discusses the process by which the IR is manipulated or massaged into the desired C source file.

3.1 Conversion Process

Figure 3.1 presents an overview of the IR conversion process. The C source file is first processed by GCC. A number of IR dumps are obtained from GCC at different stages in the compilation process, using the options described in Chapter 2. The dump files of interest to us are the post-optimization IR file (blocks file) and the alias file. The ExtractCFG framework first reads the original C source code and extracts global data from it. This global data consists of global variable declarations, comments and pre-processor statements, which are described outside any of the functions in the C code. ExtractCFG

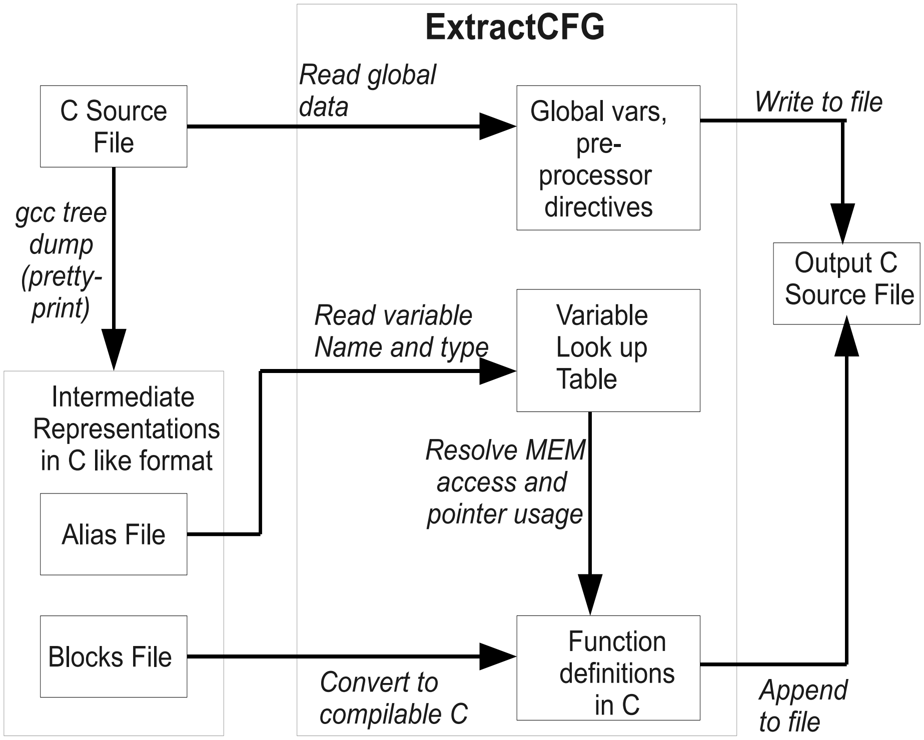

ExtractCFG
C Source File
Read global data
Global vars, pre-processor directives
Write to file
Output C Source File
gcc tree dump (pretty-print)
Read variable Name and type
Variable Look up Table
Intermediate Representations in C like format
Resolve MEM access and pointer usage
Alias File
Blocks File
Function definitions in C
Convert to compilable C
Append to file

Figure 3.1: Overview of the ExtractCFG framework.

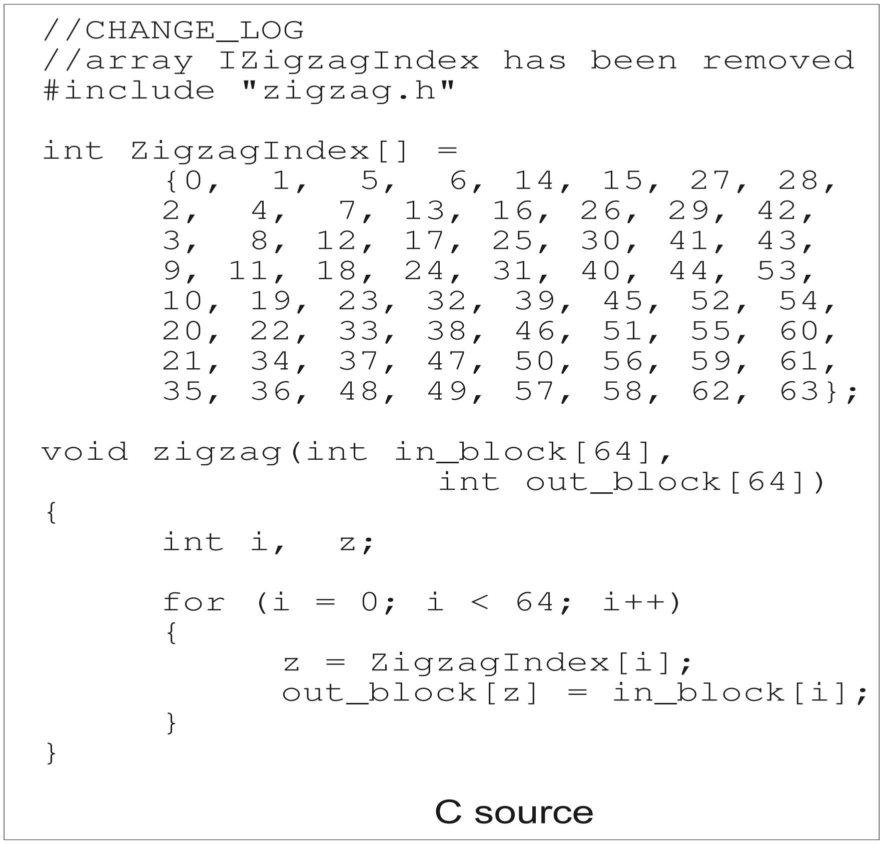
writes this data back to the global level of the destination C file. In the next step, the alias file is read to create a look-up table of the variables used in the IR file. Finally, function definitions in the IR file are manipulated to create compilable function definitions in C while retaining the CFG structure of the IR file. The variable look-up table is used in this conversion process. The C function definitions are appended to the destination C file. The converted C file is compilable and has a clearly defined basic block structure, which can be conveniently used for accurate timing back-annotation.

It is to be noted that although the GIMPLE format has its own grammar, there is no documented standard for the ‘pretty print’ format. A change in the source code of GCC, particularly in the ‘pretty-print.c’ and ‘pretty-print.h’ [1] files can lead to a change in the IR file. There are small differences in the intermediate dump files generated by different versions of GCC for the same source file with identical optimization options. The framework is thus not guaranteed to work for all C source files, or all GCC versions. However, the framework has been developed to be general, and it has been tested with source code using different C language structures and keywords. Development and testing of ExtractCFG is a continuous process, and the framework has been growing more robust over time. The framework currently works for a varied number of C source files on three different versions of GCC: version 4.1.2, version 4.1.3, and version 4.4.3.

The ExtractCFG framework is implemented as a number of Python modules. The code for the framework is written in a modular and readable fashion, in order to facilitate future modifications.

3.2 Conversion Example

In this section, we demonstrate the conversion process on a simple example, and we examine a C source file, the IR files generated from it by GCC and the final C file generated by ExtractCFG. Figure 3.2 is an example of a simple C file with one function. It is to be noted that the ExtractCFG framework can deal with multiple functions in one source file. This example

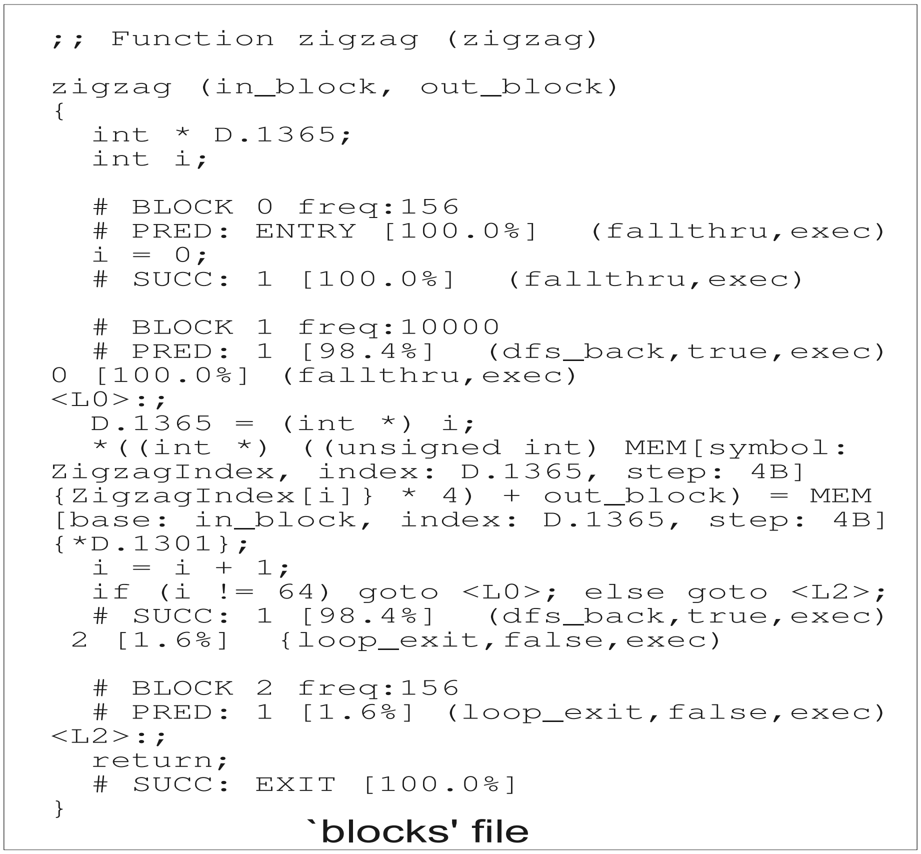


```
//CHANGE_LOG
//array IZigzagIndex has been removed
#include "zigzag.h"

int ZigzagIndex[] =
      {0,  1,  5,  6, 14, 15, 27, 28,
      2,  4,  7, 13, 16, 26, 29, 42,
      3,  8, 12, 17, 25, 30, 41, 43,
      9, 11, 18, 24, 31, 40, 44, 53,
      10, 19, 23, 32, 39, 45, 52, 54,
      20, 22, 33, 38, 46, 51, 55, 60,
      21, 34, 37, 47, 50, 56, 59, 61,
      35, 36, 48, 49, 57, 58, 62, 63};

void zigzag(int in_block[64],
                  int out_block[64])
{
      int i,  z;

      for (i = 0; i < 64; i++)
      {
            z = ZigzagIndex[i];
            out_block[z] = in_block[i];
      }
}
```

Figure 3.2: Original C file.

was chosen for its simplicity. The post-optimization IR file and the alias file generated from it are shown in Figure 3.3 and Figure 3.4 respectively.

The ‘blocks’ IR file is obviously not compilable. The variable and label names are not C compliant and the IR file contains structures like ‘MEM’, which are not supported in the C syntax. There are other, more subtle differences in the IR file, which we will discuss later in the chapter. Global information present in the original file is missing from the IR, and the `include`

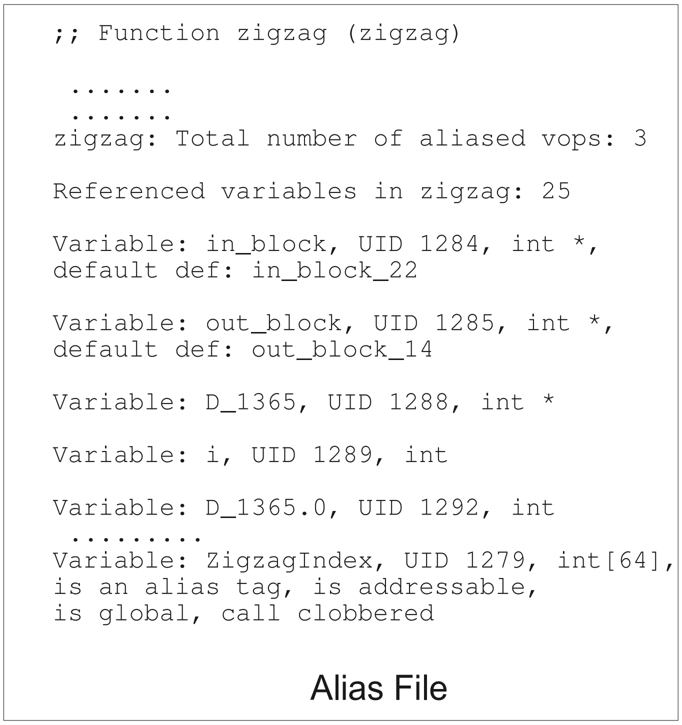


```
;; Function zigzag (zigzag)

zigzag (in_block, out_block)
{
  int * D.1365;
  int i;

  # BLOCK 0 freq:156
  # PRED: ENTRY [100.0%]  (fallthru,exec)
  i = 0;
  # SUCC: 1 [100.0%]  (fallthru,exec)

  # BLOCK 1 freq:10000
  # PRED: 1 [98.4%]  (dfs_back,true,exec)
0 [100.0%] (fallthru,exec)
<L0>:;
  D.1365 = (int *) i;
  *((int *) ((unsigned int) MEM[symbol:
ZigzagIndex, index: D.1365, step: 4B]
{ZigzagIndex[i]} * 4) + out_block) = MEM
[base: in_block, index: D.1365, step: 4B]
{*D.1301};
  i = i + 1;
  if (i != 64) goto <L0>; else goto <L2>;
  # SUCC: 1 [98.4%]  (dfs_back,true,exec)
 2 [1.6%]  {loop_exit,false,exec)

  # BLOCK 2 freq:156
  # PRED: 1 [1.6%] (loop_exit,false,exec)
<L2>:;
  return;
  # SUCC: EXIT [100.0%]
}
```

Figure 3.3: Post-optimization IR file.

statement and global variable declarations are not present. Furthermore, comments in the original code are lost. On the other hand, we can clearly see the final CFG. The edges are marked by '# BLOCK' and '# SUCC' statements. The exposed boundaries of the basic blocks allow for accurate timing back-annotation at the basic block level.

A snippet from the alias file is shown in Figure 3.4. A list of all the variables referenced by a function is present in this file. In Figure 3.4, we see

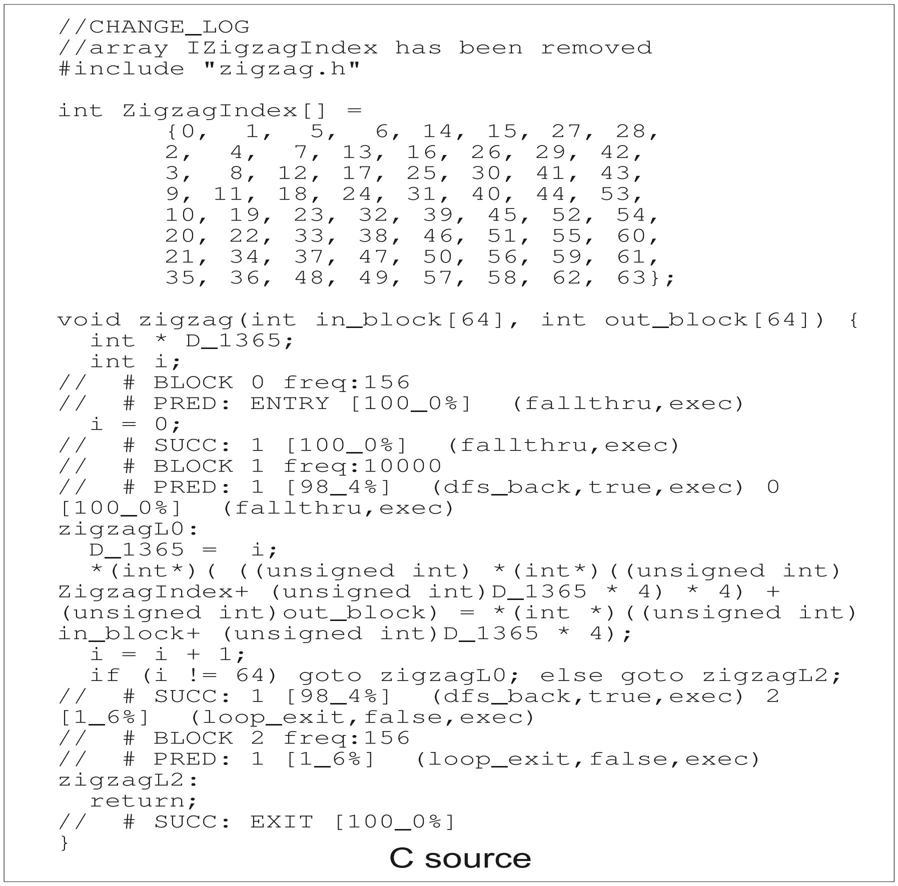


```
;; Function zigzag (zigzag)

 .......
 .......
zigzag: Total number of aliased vops: 3

Referenced variables in zigzag: 25

Variable: in_block, UID 1284, int *,
default def: in_block_22

Variable: out_block, UID 1285, int *,
default def: out_block_14

Variable: D_1365, UID 1288, int *

Variable: i, UID 1289, int

Variable: D_1365.0, UID 1292, int
 .........
Variable: ZigzagIndex, UID 1279, int[64],
is an alias tag, is addressable,
is global, call clobbered
```

Figure 3.4: Alias file.

a list of local variables, global variables and variables introduced by different stages of the compiler, for the function named zigzag. The format of this file allows for easy extraction of information to create a variable look-up table for each function.

The final output of the conversion process is a C source code file. Figure 3.5 shows the converted C code obtained from the C source code shown in Figure 3.2. The converted C code has the same functionality as the original code and has basic block boundaries clearly annotated as comments. This code



```
//CHANGE_LOG
//array IZigzagIndex has been removed
#include "zigzag.h"

int ZigzagIndex[] =
       {0,  1,  5,  6, 14, 15, 27, 28,
       2,  4,  7, 13, 16, 26, 29, 42,
       3,  8, 12, 17, 25, 30, 41, 43,
       9, 11, 18, 24, 31, 40, 44, 53,
       10, 19, 23, 32, 39, 45, 52, 54,
       20, 22, 33, 38, 46, 51, 55, 60,
       21, 34, 37, 47, 50, 56, 59, 61,
       35, 36, 48, 49, 57, 58, 62, 63};

void zigzag(int in_block[64], int out_block[64]) {
  int * D_1365;
  int i;
//  # BLOCK 0 freq:156
//  # PRED: ENTRY [100_0%]  (fallthru,exec)
  i = 0;
//  # SUCC: 1 [100_0%]  (fallthru,exec)
//  # BLOCK 1 freq:10000
//  # PRED: 1 [98_4%]  (dfs_back,true,exec) 0
[100_0%]  (fallthru,exec)
zigzagL0:
  D_1365 =  i;
  *(int*)( ((unsigned int) *(int*)((unsigned int)
ZigzagIndex+ (unsigned int)D_1365 * 4) * 4) +
(unsigned int)out_block) = *(int *)((unsigned int)
in_block+ (unsigned int)D_1365 * 4);
  i = i + 1;
  if (i != 64) goto zigzagL0; else goto zigzagL2;
//  # SUCC: 1 [98_4%]  (dfs_back,true,exec) 2
[1_6%]  (loop_exit,false,exec)
//  # BLOCK 2 freq:156
//  # PRED: 1 [1_6%]  (loop_exit,false,exec)
zigzagL2:
  return;
//  # SUCC: EXIT [100_0%]
}
```

C source

Figure 3.5: Converted C code file.

can be correctly back-annotated and then used for host compiled simulation. The basic block boundaries are clearly demarcated with the `#BLOCK` and `#SUCC` statements. Back annotations can be made in the code at the beginning of each basic block.

A minor disadvantage of the converted C code is that it is significantly longer than the original code and less readable. Comments inside functions are also lost during conversion. This is not a major issue because development and refinement is performed on the original C code and this version of the code is used only for timing back-annotation and simulation.

3.3 IR to C Conversion

In this section, we will describe the different steps of the IR conversion process in more detail. The ExtractCFG framework uses the original C code, the alias file and the IR dump to generate the final C code. As discussed previously, this is done in three major steps: extracting global data, creating a variable look-up table and generating the final C code.

3.3.1 Extracting Global Data

The framework first reads the original C file and then extracts all the data that is present outside the function definitions. This data consists primarily of comments, include statements and global variable declarations, and is copied directly to the final C file. The problem of global data not being present in the 'blocks' IR file is hence solved.

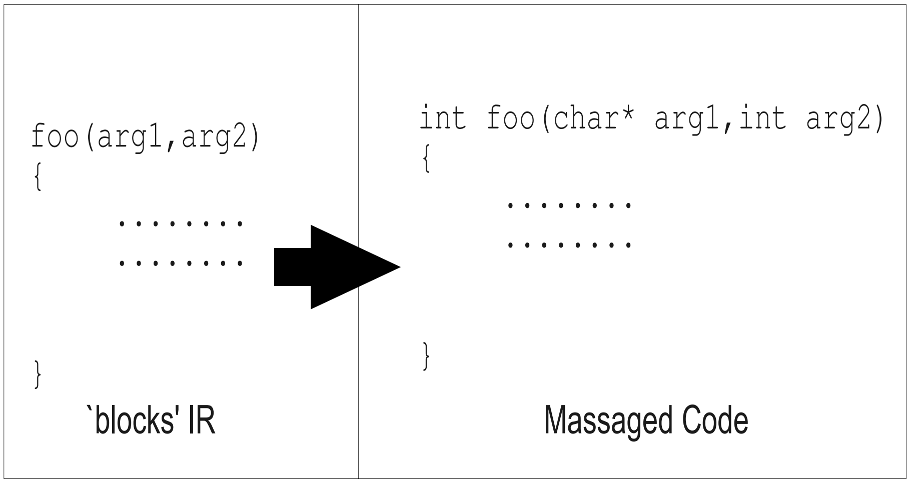
The IR file also does not have the return type and parameter type of functions. This information is extracted from the original C code for each of the functions contained therein. The function parameter and return data are stored as members in an instance of a `fn` class internal to ExtractCFG. Each function in the source code has an object of `fn` class created for it.

3.3.2 Creating a Variable Look-up Table

Type information about global variables declared in header files and some compiler-introduced variables are present neither in the original C code nor the final IR file. This information is required for conversion of the IR file to compilable C code. The 'alias' dump file has information about the variables referenced by each function. This includes global variables, user-defined local variables, and local variables introduced during the GCC compilation process. The alias file is read and a variable look-up table is created by using the dictionary data structure in Python. This table maps each variable to its type information. One look-up table is created for each function and stored as a member in the `fn` object.

3.3.3 C Code Generation

The IR file has a number of operations performed on it to massage or manipulate its contents into compilable C code. All these operations are performed in a single pass of the 'blocks' IR file in order to increase ExtractCFG's execution speed. The `fn` objects are initialized with values from the C source



```
foo(arg1,arg2)
{
      ........
      ........


}
```

```
int foo(char* arg1,int arg2)
{
      ........
      ........


}
```

Figure 3.6: Inserting function parameters.

file and the alias file before the IR file is processed. Code conversion of the IR is done line-by-line and each processed line is appended to the target C file.

Some of the major operations in manipulating the IR file are described below. All these operations are performed on each line of the IR code, as applicable.

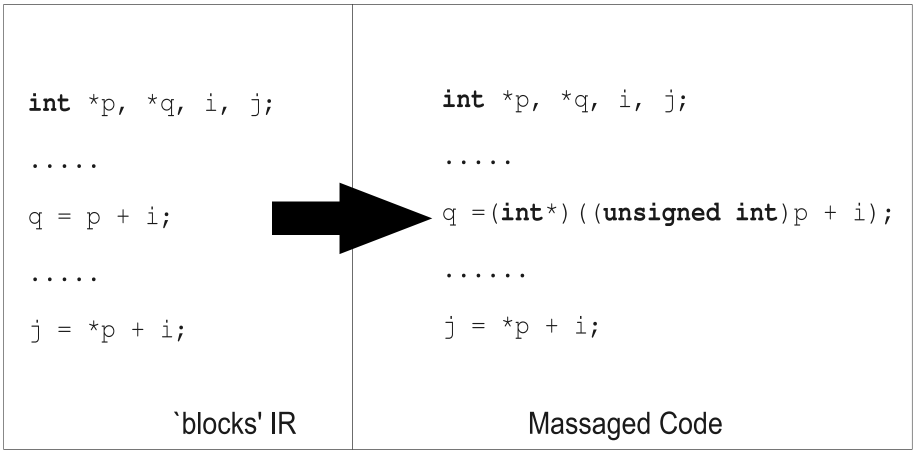
3.3.3.1 Inserting Function Parameters

In the the ‘blocks’ file, the return value and parameter types of functions are inserted into the first line of function definitions. These values are obtained from the `fn` class instance for each function. Figure 3.6 shows an example of this process. The definition of function `foo` in the IR file does not have the return type specified. Likewise, the type information for its parameters `arg1` and `arg2` is also not present. This information is inserted into the function definition according to C syntax rules. The function definition of `foo` now specifies that its return value is an integer and that its arguments are a character pointer and an integer.

3.3.3.2 Handling Pointer Arithmetic

There is no concept of C style pointer arithmetic in the IR file. For example, in the IR, adding the number 4 to an integer pointer is supposed to increment the address it points to by just 4 bytes. In C, this expression would instead advance the address by 16 bytes, assuming a 4 byte integer. Whenever a pointer occurs in an arithmetic operation in the IR, ExtractCFG casts the pointer into an `unsigned int`. The result of the arithmetic expression must be cast back to the original pointer type. Pointers and their data types are identified using the look-up table described earlier. This step must also distinguish between pointers and dereferenced pointer values.

In the example shown in Figure 3.7, the pointer `p` is cast to an unsigned integer when it occurs in an arithmetic expression. The result of the operation is cast into an integer pointer, since `p` is a pointer to an integer in the original code. The second arithmetic expression, which involves the dereferenced value of `p`, is left alone, since `*p` refers to an integer value.



```
int *p, *q, i, j;
.....
q = p + i;
.....
j = *p + i;
```

```
int *p, *q, i, j;
.....
q =(int*)((unsigned int)p + i);
......
j = *p + i;
```

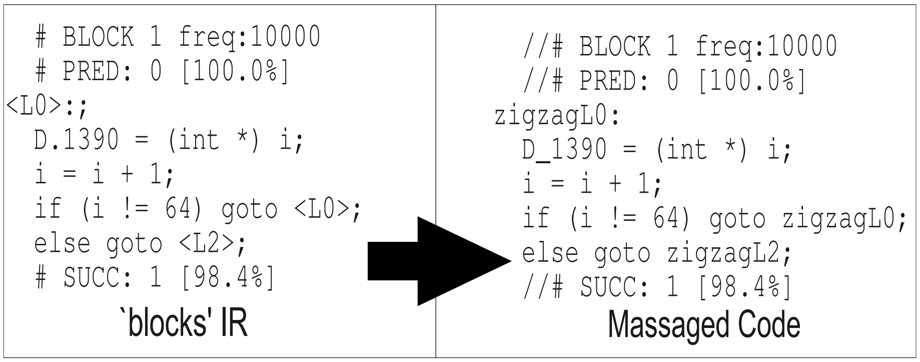
Figure 3.7: Pointer arithmetic example.

3.3.3.3 Creating C Style Labels

Labels are used extensively in the GIMPLE format as a target of `goto` statements. Labels are converted by ExtractCFG to a C compilable form, which includes joining the function name and the label number. The function name is appended so that all labels in one source file have a unique name, even if they occur in separate functions. In Figure 3.8, we see an example of this process. This example shows how the function name `zigzag` is added to the label number `L0` to get the label name `zigzagL0`. Special characters are also removed from the label name.

3.3.3.4 Managing CFG Data

CFG data is present in the 'blocks' file in the form of `# BLOCK`, `# PRED # SUCC` statements. This data is not compilable, but it is necessary for marking



```
  # BLOCK 1 freq:10000
  # PRED: 0 [100.0%]
<L0>:;
  D.1390 = (int *) i;
  i = i + 1;
  if (i != 64) goto <L0>;
  else goto <L2>;
  # SUCC: 1 [98.4%]
```

```
  //# BLOCK 1 freq:10000
  //# PRED: 0 [100.0%]
zigzagL0:
  D_1390 = (int *) i;
  i = i + 1;
  if (i != 64) goto zigzagL0;
  else goto zigzagL2;
  //# SUCC: 1 [98.4%]
```

Figure 3.8: Label and CFG conversion.

the edges where timing annotation will take place. At most edges, the CFG data also contains statistics about the relative frequency of execution of a block, and the percentage of times a conditional branch is taken. So we leave this information in the final C file, and just comment it out using C-style comments.

In Figure 3.8, we see that the CFG statements are present in the final code, but they have been commented out in the generated code. Hence, basic block information is preserved in the final file without losing compilability.

3.3.3.5 Memory Accesses

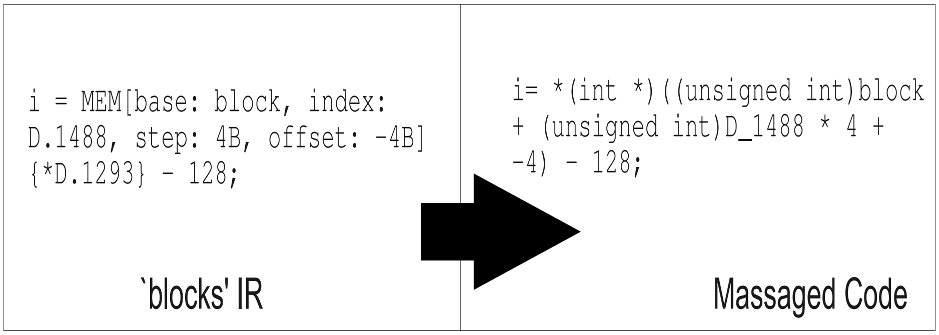
Sequential memory accesses are represented in the 'blocks' file using a special `MEM` syntax. This MEM syntax has the following structure:

`MEM[symbol: e1, base: e2, index: e3, step: d1, offset: d2] {Prev_Val}`, where e1, e2, e3 might be pointers or pointer arithmetic expressions, whereas d1 and d2 are integers. All parameters are optional, but one of the three parameters (symbol, base or index) must be present in the MEM structure. The field Prev_Val is the variable that the compiler replaced with the MEM structure at a previous compilation stage. The address that the MEM structure refers to is decoded as:

```
(e1 + e2 + e3 * d1 + d2) .
```

If d1 is not specified it defaults to 1. All the other parameters default to zero. The value of this expression is an address. The data type of the value that is to be accessed at this location is still unknown. The data type of Prev_Value is retrieved from the variable look-up table and the calculated address is cast as a pointer to that data type. Then, a dereferencing operator (*) is added to access the value at that memory address. This is a complicated conversion operation, since the expressions e1, e2, e3 are often often arithmetic expressions involving pointers, which need to be handled as discussed earlier.

In Figure 3.9, we see an example of this process. In this example, the values in the block and index fields are pointers. These values are cast as unsigned integers before calculating the memory address. The values in the step field is 4B, which means four bytes. The trailing character is stripped from this field before performing the address calculation. The offset field is processed in the same way. The type of variable accessed from the calculated location is obtained from the term *D.1293. For this example, the data type of D.1293 is an integer pointer. The value *D.1293 that was replaced by the



```
i = MEM[base: block, index:
D.1488, step: 4B, offset: -4B]
{*D.1293} - 128;
```

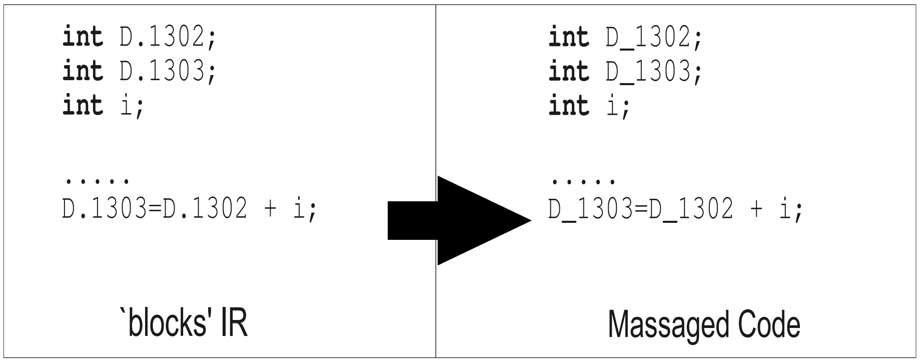
```
i= *(int *)((unsigned int)block
+ (unsigned int)D_1488 * 4 +
-4) - 128;
```

Figure 3.9: MEMORY access example.

MEM structure is an integer. Therefore, the calculated address is cast as an integer pointer and the the dereferencing operator is inserted before it.

3.3.3.6 Variable Renaming

The variables in the ‘blocks’ IR are not all C compliant. This is true of all compiler introduced variables. These variables are converted to standard C variables. Figure 3.10 shows an example of this. The variables are converted to syntactically correct C variables by replacing special characters with an underscore (_) character. This step is performed towards the end of the conversion process for each line to allow other conversion steps to correctly read the variable look up table. Figure 3.10 shows an example of this process. The variable names in the IR have period(.) characters in their names, which is not legal C syntax. The variables `D.1302` and `D.1303` are renamed to `D_1302` and `D_1303`.



```
int D.1302;
int D.1303;
int i;

.....
D.1303=D.1302 + i;
```

```
int D_1302;
int D_1303;
int i;

.....
D_1303=D_1302 + i;
```

Figure 3.10: Variable renaming.

3.3.3.7 Miscellaneous

There are a number of other tweaks that must be done in order to generate compilable code. A few are deleting duplicate variable replacements, dealing with other tree structures like MIN_EXPR, MAX_EXPR, ABS_EXPR and ensuring that keywords or functions names are not confused as variables. After all the conversions have been performed, the final result of this process is a compilable C file that fully conforms to C syntax while maintaining the exact semantics of the original source code (see Figure 3.5). The final generated C file can hence form the basis for host-compiled simulation and further timing or power back-annotation through other tools.

Chapter 4

Results and Verification

The success of the ExtractCFG Framework can be judged on the basis of the different kinds of C programs it can handle, and on how closely the output C file matches the control flow graph of the post-optimization GIMPLE tree and the final assembly code. The framework has been extensively tested and it behaves as desired for a large set of C source files and target platforms. This chapter discusses the testing and verification methodology used for the framework, as well as the observed results.

4.1 Test Setup

The framework was tested with a number of input C files. Some of these files were written expressly for the purpose of testing the framework, and some files were from real world examples. Three different GCC versions were tested for three different targets.

4.1.1 GCC versions and Target Platforms Tested

The format of the intermediate representation dump files changes slightly with each version of GCC. Due to its architecture, ExtractCFG needs only minor modifications in order to support newer GCC versions. ExtractCFG was tested successfully on the following GCC versions and packages:

```
GCC VERSION 4.4.3 for Linux on Intel i386

GCC VERSION 4.1.2 for Linux on Intel i486

GCC VERSION 4.2.3 for Linux on ARM
```

4.1.2 Ad-Hoc Testing

A set of artificial C files were written specifically for the purpose of testing the framework. There are 16 such test cases, with 4 to 5 variations of each test case. As discussed earlier, the IR dump files do not follow any fixed standard. Hence, extensive testing is required to make ExtractCFG support a wider variety of C input files. All common C constructs and data structures were tested in this phase, such as `for` loops, `while` loops, `do while` loops, `switch case` along with 1D and 2D arrays, and `structs`. Complicated tests were also constructed by combining some of these elements. The functionality of the generated code is tested and found to be the same as the original code in all these test cases.

4.1.3 Real World Examples

Two real world examples were also used to test the framework. These examples are implementations of a JPEG encoder algorithm and of a Fast Fourier Transform algorithm.

JPEG is a standard algorithm for lossy compression of digital images.

The JPEG encoder code has 8 source files and 7 header files with a total of 1329 lines of code. The JPEG algorithm involves a number of memory buffers, complicated program flow, extensive pointer usage and the use of file IO. ExtractCFG was invoked on all the 8 source files, and 8 new C source files were obtained. These new source files along with the previously existing headers were successfully compiled. The functionality of the converted code was tested by comparing the JPEG files it generated against the JPEG files generated by the original code. The files were found to be identical for all four input bitmap files tested.

Fast Fourier Transform (FFT) is a commonly used algorithm in many Digital Signal Processing (DSP) systems. The FFT implementation we use is not a very efficient one and hence its IR after compiler optimization should be significantly different from the original source. The FFT consists of one source file with 195 lines of code. There is a fair amount of floating point arithmetic involved, as well as terminal IO. The CFG is relatively simple because of the sequential nature of the FFT algorithm. The generated code was found to be identical in functionality to the original code for all 5 sets of inputs tested

4.2 Functional Validation of ExtractCFG

The ExtractCFG framework can be said to to meet its design specifications if the functionality of the generated code is identical to that of the original code and if the CFG of the IR file and the CFG of the generated C code are the same. We validated the functionality of the generated C code by running both the original and the converted C code for identical test cases and comparing their respective outputs. As mentioned earlier, the functionality of the original C code was observed to remain unchanged after conversion for all the test cases.

The CFG of the generated C file is compared with that of the ‘blocks’ intermediate file. This comparison was done manually for all the test cases. This validation is to check if the massaging code has in some way altered the CFG. No discrepancies were recorded for any of the input files.

4.3 Performance and Code Size Comparison

In this section we compare the code size, and execution times of the converted code to that of the original one. Table 4.1 compares the lines of code of the two versions of code for some of the C files in the test cases. It is observed that the newer version of code has significantly more lines of code, with an average increase of 68%. As expected, this reduces the readability of the generated C code, but this is not a major concern.

The size of the executable files is compared to check if it increases when compiled from the generated C code due to the conversion process. The results of this comparison are shown in Table 4.2. As expected, there is no appreciable change. In fact, the executable sizes are identical for the jpegencoder example.

The execution times of the new and old versions of code are also compared, when they are compiled for the host machine. The programs were run

Table 4.1: Comparison of lines of code.

C File	**Original Code**	**Converted Code**
dct.c	129	310
file.c	23	60
huffencode.c	668	1242
jpegencoder.c	39	51
quantize.c	37	84
read.c	24	62
ReadBmp_aux.c	248	249
zigzag.c	35	53
fft.c	195	243

Table 4.2: Comparison of executable size in bytes.

Test Case	**Original Code**	**Converted Code**
fft	6542	6510
jpegencoder	17659	17659

10,000 times in a loop to record the execution times shown in Table 4.3. A very small increase in execution time for the newer version of the code is observed. The conversion process therefore does not introduce any appreciable overhead into the code, and will not slow down the host compiled simulation. The back-annotation of timing will however slow down the code by a more significant amount. Depending on the granularity level of annotation we can expect reasonably fast host-compiled simulation.

Table 4.3: Comparison of execution speeds for 10,000 iterations.

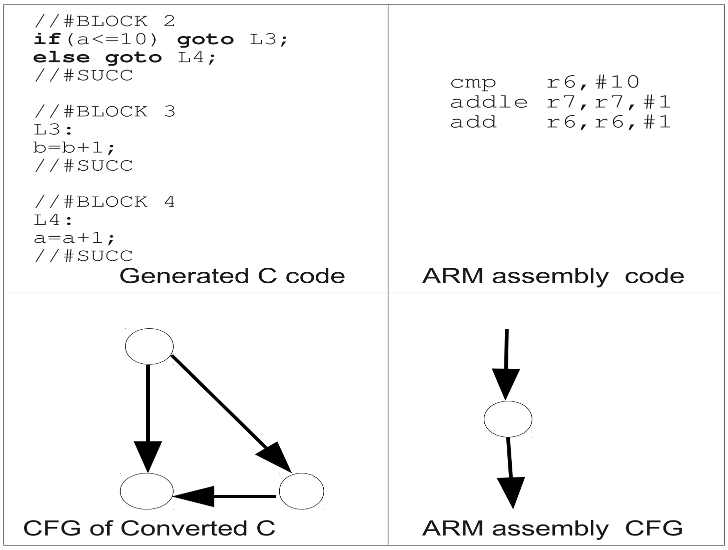
Test Case	Original Code	Converted Code
fft	10.658 sec	10.665 sec
jpegencoder	25.953 sec	26.458 sec

4.4 Usability for Back Annotation

The generated C file can be said to be a suitable candidate for back annotation if its CFG reflects that of the final assembly code. For our experiments, we manually obtained the CFG of the target assembly file by looking at jump statements and their associated labels in the assembly code. The generated C file is regarded as a good candidate for back-annotation at the block level if timing annotations in the C file can account for every execution path in the generated assembly. Assembly files were generated for the Intel x86 and ARM platforms for four test cases, and were analyzed for this purpose.

4.4.1 Intel x86

Generally, the sequential order of basic blocks in the assembly code are observed to differ from that of the generated C file. This is because of the block reordering pass in the RTL back-end of GCC. However, upon inspection of the actual execution paths (edges of the CFG) it can be observed that the actual flow graphs are not affected by this reordering, i.e. the CFG of the assembly code is identical to that of the generated C file. The C file created by ExtractCFG is therefore suitable for inserting timing annotations.



```
//#BLOCK 2
if(a<=10) goto L3;
else goto L4;
//#SUCC

//#BLOCK 3
L3:
b=b+1;
//#SUCC

//#BLOCK 4
L4:
a=a+1;
//#SUCC
```

Generated C code

```
cmp    r6,#10
addle  r7,r7,#1
add    r6,r6,#1
```

ARM assembly code

CFG of Converted C

ARM assembly CFG

Figure 4.1: Comparison of ARM assembly and C file CFG.

4.4.2 ARM Targets

In the case of ARM targets, even after taking block reordering into account, the CFG obtained from an ARM assembly sometimes differs from the CFG of the C file. An example of this is shown in Figure 4.1. In the example, we see that three basic blocks in the C file are combined into one basic block in the ARM assembly code. This is due to the fact that some ARM opcodes have a conditional execution option. These opcodes might or might not execute depending on the value of certain status flags. The '`addle r7,r7,#1`' opcode in the example executes only if the '`le (less than)`' conditional flag is true. However this opcode will take the same time to execute in both cases.

Overall, the functionality of the generated C file is identical to that of the assembly, but it needs a different approach to timing annotation. The block of assembly code shown will always execute in sequence. Since the outcome of the comparison statement does not affect execution time on the ARM, only one timing annotation needs to be inserted in the C code at the beginning of block 2. We can conclude that, despite the changes in the CFG of the final ARM assembly code, the generated C file is still a good candidate for back annotation even on the ARM platform.

Chapter 5

Summary and Conclusions

The ExtractCFG project was started with the aim to enable timing annotation in high level code, while at the same time being able to take into account compiler optimizations. This aim has been achieved by converting a dump file of a GCC intermediate representation of the code back into compilable C form. The C file generated by ExtractCFG can be used for host-compiled simulation of code execution on the target platform. Since the framework uses hardware independent intermediate representations as inputs, it can be used for a variety of target platforms, with little or no modifications.

This project had a steep learning curve. A good understanding of the optimization and IR conversion stages of GCC was required. The lack of documentation or a standard format for the 'pretty print' representation meant a trial-and-error approach and extensive gathering of data from GNU's online forums. The inconsistency of this format also meant that complex regular expressions were needed to handle corner cases. The structure and coding style of ExtractCFG was still kept as modular and easily maintainable as possible.

The ExtractCFG project has passed the 'proof-of-concept' stage. It has been tested on a range of artificial and real-world test cases for a variety of compiler versions and target platforms. In all cases, the framework has been shown to produce correct code with little to no execution time overhead. Future work can be done to extend it to support different GCC versions. Furthermore, additional real-world applications need to be tested on the framework to improve its stability.

The possibility of extending this framework to C++ and other languages can also be explored. Since the GIMPLE representation and the 'pretty print' format are independent of the front-end language, it is theoretically possible to use the same approach for other languages, and get a C file with the desired CFG.

Bibliography

[1] GCC releases. http://gcc.gnu.org/releases.html.

[2] A.V. Aho, M.S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[3] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007.

[4] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International Euro-Par Conference on Parallel Pro- cessing*, pages 1298–1307, 1997.

[5] D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis, Verification*. Springer, September 2009.

[6] A. Gerstlauer. Host-compiled simulation of multi-core platforms. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*, Washington,DC, USA, June 2010.

[7] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara. Specify-explore-refine (SER): From specification to implementation. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, USA, July 2008.

[8] B.J. Gough. *An Introduction to GCC for the GNU Compilers gcc and g++*. Network Theory Ltd, 2004.

[9] L.J. Hendren, C. Donawa, M. Emami, G.R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 457 in Lecture Notes in Computer Science, pages 406–420. Springer-Verlag, August 1992.

[10] Y. Hwang, S. Abdi, and D. Gajski. Cycle approximate retargettable performance estimation at the transaction level. *Proceedings of Design, Automation and Test in Europe (DATE)*, March 2008.

[11] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.

[12] D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *GCC Developers' Summit*, pages 181–193, Ottawa, Canada, May 2003.

[13] D. Novillo. GCC - an architectural overview, current status and future directions. In *Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[14] P. Puschner. Worst-case execution time analysis at low cost. *Control Engineering Practice*, 6:129–135, 1998.

[15] P. Razaghi and A. Gerstlauer. Host-compiled multicore RTOS simulator for embedded real-time software development. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Grenoble, France, May 2011.

[16] G. Schirner, R. Dömer, and A. Gerstlauer. *Hardware-dependent Software: Principles and Practice*. Springer, 2009.

[17] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Design Automation Conference DAC*, pages 290–295, 2008.

[18] R.M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. Retrieved from http://gcc.gnu.org/onlinedocs/gccint .pdf.

[19] R.M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection: for GCC version 4.6.1 (GCC)*. Retrieved from http://gcc. gnu.org/onlinedocs/gcc-4.6.1/gcc.pdf.

[20] M. Streubühr, C. Haubelt, and J. Teich. System level performance simulation for heterogeneous multi-processor architectures. *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, January 2009.

[21] Wang Z. and A. Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. *Design Automation Conference (DAC)*, July 2009.

[22] Wang Z. and A. Herkersdorf. Flow analysis on intermediate source code for wcet estimation of compiler-optimized programs. *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.