FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

Host Compiled Simulation for Timing and Power Estimation

Gaurav Kukreja

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

Host Compiled Simulation for Timing and Power Estimation

Author:	Gaurav Kukreja
Supervisor:	Prof. Michael Gerndt
Advisor:	Dr. Josef Weidendorfer
Advisor:	Mr. Bo Wang
Submission Date:	15th October, 2014

I confirm that this Master's Thesis is my own work and I have documented all sources and materials used.

Munich, 15th October, 2014 Gaurav Kukreja

Abstract

Simulation is a useful technique for Hardware Software Co-development. It is performed at various levels of abstraction to serve different purposes. Instruction Set Simulation is the lowest level of abstraction where the processor pipeline is simulated in detail, to allow hardware developers to test their modifications and evaluate the impact on performance. At higher levels of abstraction, simulation provides developers with a tangible environment for early software development. The focus of this project is on simulation for performance estimation, namely, estimation of time and power consumed in running a benchmark application on a target processor.

While Instruction Set Simulators are known to be highly accurate, they are difficult to develop and slow to execute because of the level of detail they address. Host Compiled Simulation is a technique to accelerate performance estimation with negligible impact on accuracy. The idea is to instrument[1] the source code, by taking into consideration the behaviour of the target processor. The instrumented source code is compiled and run on the Host Machine. The technique relies on the assumption that performance of each basic block[2] in the binary code can be accurately estimated on a certain processor by emulating the pipeline. Other aspects that affect performance, like resources spent in memory access can be accounted for, and a fairly accurate estimate of the time and power consumed can be estimated.

In this project, a tool to perform Host Compiled Simulation was developed. This thesis discusses the state-of-art in simulation. It explains the approach used to develop this tool. The results showing accuracy of estimations from this approach are presented.

[1] Instrumentation is a technique to modify the source code of an application in order to collect statistics at run-time. This may be used to measure performance of the application, or diagnose errors.

[2] A basic block in a program is a series of instructions which are executed sequentially. The basic block does not contain branch instructions.

Contents

1 Introduction

1.1 Simulation

Simulation is the technique to imitate the behaviour of a system. It is generally used when developing or working with the real system is expensive or difficult.

Simulation is widely used in Hardware Software Co-development. The behaviour of a processor is simulated, and benchmarking applications are run. It allows Hardware Developers to analyse and validate the performance impacts of design decisions at early stage of hardware development. This saves the crucial effort and cost involved in the painstaking process of fabricating hardware at each milestone.

The popular approach used for simulation in the industry today is called Cycle Accurate Simulation (CAS). In CAS the processor micro-architecture is simulated in great detail. Each stage of processor pipeline is simulated along with other building blocks of the processor like Cache Memory and Branch Prediction Units. This approach provides cycle accurate estimates of performance. However, CAS is difficult to develop and slow to execute because of the amount of details that are simulated. Long running software benchmarks are used to accurately estimate bottlenecks in the system, and CAS is not suited for this due to the slow execution speed.

In this research, a technique for fast simulation of processors has been explored. The aim is to extract accurate estimate of number of cycles spent and power consumed by a processor in running a software benchmark.

1.2 Related Work

Research in this area has focussed on following approaches.

1.2.1 Sampling Based Approach

Sampling is an approach used in statistical analysis. Small, yet representative samples are chosen from a vast amount of data. These samples are analysed in detail, and the results are interpolated to gather information about the entire data set.

In this approach, the application is mostly run using Functional Simulation, and some samples are executed using the detailed CAS. The number of cycles spent in execution of the samples is calculated, and the number of cycles spent in executing the entire pipeline is estimated by interpolating.

This approach provides considerable speed up compared to CAS, however accuracy of the estimation is highly dependent on how the samples are chosen. Also, developing this technique is difficult, since CAS is used.

1.2.2 Host Compiled Simulation

Host Compiled Simulation is based on the approach of Source Code Instrumentation (SCI). SCI is the process of modifying source code to collect performance statistics and generate trace information during run-time.

When an application is run on a processor, most of the time is spent in

- Execution of the instructions in the processor pipeline, and
- Fetching data from the memory.

If the number of cycles spent in each of these phases can be accurately estimated, the total number of cycles spent in running the application can be calculated. In this approach, the source code is instrumented to do this. The instrumented source code is compiled for and run on the Host Machine (the machine where the simulation is run) and hence the name, Host Compiled Simulation.

1.3 Focus

The focus in this research is to develop a tool to perform Host Compiled Simulation of a processor. The tool should be able to automatically instrument a given source code. The instrumented source code will be compiled and run, and accurate estimates of the performance will be reported.

1.4 Thesis Outline

Chapter 2 covers some basic background knowledge that will be helpful in understanding the project. This chapter can be skipped by the knowledgeable readers. References to appropriate sections of the chapter is provided in the next chapters. The readers may choose to revisit these concepts if needed.

Chapter 3 presents an overview of the technique. A flow chart explaining the approach is described and the challenges in each stage are briefly illustrated using examples.

Chapter 4 deeply explains each stage of the technique in further detail. The challenges, and how they are tackled is discussed. It also sheds some light into the limitations of the tool and how it can be enhanced.

Chapter 5 presents a deep analysis of the results. The test setup is explained, and the accuracy of results from this technique is demonstrated.

2 Background

This chapter covers background knowledge that may be required to understand the project. The topics briefly covered in this chapter are as follows.

- Basics of Computer Architecture: Pipeline, Caching, Branch Prediction.
- Details of ARM Cortex A5 Processor
- Execution of Bare Metal Applications on ARM
- Cross-Compiling using GCC
- Compiler Optimization
- Extracting Debug Information using GDB

Basic understanding of these topics is needed to understand this thesis. Readers may choose to skip this chapter. Further in the thesis, appropriate references are made to sections in this chapter. Reader may choose to revisit these sections if required.

2.1 Basics of Computer Architecture

2.1.1 Pipeline Execution

Most modern processors use a multi-stage pipeline architecture for the execution unit. Execution of an instruction can be divided into multiple stages as follows:

- Instruction Fetch
- Instruction Decode
- Data Fetch
- Execute
- Write Back

This is only a general representation of the break down, and different processors use a slightly modified break down. The task performed in each of the stage is self-explanatory and usually, one clock cycle is spent in executing each stage. The execution unit of the processor is divided into separate units for each stage of the instruction execution.

This allows for parallel execution of instructions. At the beginning of execution, the pipeline is empty and the first instruction is fetched from the memory. While this instruction is being decoded the subsequent instruction is being fetched and so on. This helps in achieving a performance of nearly 1 Cycle Per Instruction (CPI), when no branching instructions are encountered and all data is available in registers.

This performance may not always be achievable due to interlocking between instructions. Interlocking occurs when subsequent instructions have control or data dependency. If an instruction depends on the result of the previous instruction, the pipeline must be stalled unless the result is made available.

This performance is also hindered by conditional branching instructions. A simple *if-then-else* branch is implemented as a compare instruction followed by a conditional branch instruction in the binary code. Depending on the result of the compare instruction, the branch instruction is either executed or ignored. The processor uses Branch Prediction Algorithms to predict the next instruction that will be executed and already feeds the instruction to the pipeline. Incorrect prediction leads to Pipeline Flush, which results in a wastage of a couple of cycles.

2.1.2 Cache Hierarchy

Memory access is a critical bottleneck in modern computers. Main memories used today take over hundreds of cycles to fetch data, while the processor halts execution waiting for the data to be delivered. This leads to considerable waste of resources.

To optimize this, processors use a hierarchy of low-latency cache memories. The level 1 or L1 Cache is closely coupled with the processor. It has very low-latency for read and write operations, but is much smaller in size compared to the main memory. L2 Cache has higher latency than L1 Cache, but is much faster compared to main memory. Further levels of caching may be used, but it has been observed that the best performance is achieved by using 2 or 3 levels of caching.

Different techniques can be used to store and lookup data in a cache. The most popular technique is to use an N-way Set Associative Cache which has been explained graphically in figure [TODO]. The cache memory is divided into *n-ways* of equal size. Each *way* has a number of fixed size cache lines which can be indexed. Each entry in the cache contains a *tag*, the *data* and *flags* which tell the validity of the data.

When data on an address is requested by the processor, the address is divided into *tag* and *index*. The data can only reside in one of the cache ways in the cache line indexed by *index*. The cache controller looks for valid data in the cache lines, which has the same *tag* as the *tag* derived from the address. If such a data is found, a cache hit occurs and the data is made available to the processor. If the data is not found, a cache miss occurs and lookup is carried out in further levels of cache. If none of the caches contain the data, it must be fetched from the memory. At this point, the data along with the tag is stored in the cache. If the same data is requested again, it is now available in cache and can be delivered with little latency.

For a write operation, different policies can be used. On receiving a write operation request from the processor, a cache with Write-Through policy will immediately write the data back to the main memory. The processor must remain in an idle state until this write operation is completed. In Write-Back policy however, the data will only be written to the cache, and it will be flagged as *dirty* data. The data will be written to the main memory later, and the processor does not need to wait, thereby improving performance. Most processors today use Write-Back Policy.

Separate caches are usually used to store Instruction and Data, as the access pattern for the two is generally quite different.

Obviously, there is a limit to the amount of data that can be stored in the cache, and hence the cache controllers use a cache replacement policy to replace data which is less likely to be used again. Popular replacement policies are Least Recently Used and Pseudo Random Replacement.

The memory management unit also tries to analyse the memory access pattern, and prefetches the data that the processor could use in future. This reduces the number of cache misses, and optimizes performance.

This is just a basic understanding of how caches work, in a computer with a single processor. In multi-processor systems each processor may have a private L1 Cache. Different processors may try to read and write to the same address simultaneously leading to race conditions. The cache controllers must ensure data coherency. This topic is not discussed here, because the focus for this research is on single-processor systems.

2.1.3 Branch Prediction

The issue of pipeline flushing in presence of branching instructions was discussed in section 2.1.1. Processors use Branch Prediction Algorithms to improve performance by trying to predict the result of the conditional branch instruction. On fetching a branch instruction, the Branch Prediction Unit tells the Instruction Fetch unit in the pipeline the address of the next instruction to be fetched according to the prediction. If the prediction holds true, a few clock cycles are saved on execution. If the prediction was incorrect, the pipeline is flushed.

The general policy for Branch Prediction utilizes a track record of all branch instructions encountered previously. It uses a state machine (figure [TODO])for each branch instruction, and maintains the state of the machine in the table using 2 bits as follows.

- 00 - Strongly Not Taken (SNT)
- 01 - Weakly Not Taken (WNT)
- 10 - Weakly Taken (WT)
- 11 - Strongly Taken (ST)

When in SNT and WNT state, the Branch Prediction Unit predicts that the branch will not be taken. Similarly, in WT and ST states it predicts that the branch will be taken. When a branch instruction is seen for the first time, the state is set to SNT. The state changes depending on whether the prediction was correct or incorrect.

2.2 ARM Cortex A5 Processor

ARM Cortex A5 has been used as the target processor to analyse the results in this research. In this section, features of the processor have been briefly described. Only the details relevant for understanding the thesis have been covered.

2.2.1 Processor Pipeline

The Cortex A5 has an 8 stage processor pipeline. Each stage in the pipeline takes 1 cycle to execute. Interlocking may occur between instructions, which may lead to pipeline stalls.

2.2.2 Cache Hierarchy

The Cortex A5 uses separate L1 cache for data and instructions. The size of the L1 Cache is configurable from 2 KB to 64 KB. On the target processor used for testing, the 32 KB of cache memory was used for Data and Instruction Caches. Each cache line has a fixed size of 32 Bytes.

The Data Cache is 4-way Set Associative Cache, with Write-Back Policy. The Instruction cache is a 2-way Set Associative Cache. The replacement policy used is Pseudo Random Replacement.

The L2 Cache is a unified Cache for data and instruction. It is 256 KB in size and is a 16-way Set Associative Cache with 32 Byte cache line lenght. It also uses Write-Back Policy, and Pseudo Random Replacement strategy.

The Cache controller ensures that each data is only cached in one of the caches. When a data is fetched from main memory, it is stored in the L1 Cache. When a data needs to be evicted from L1 Cache, it is stored in L2 Cache. In case of a L2 Cache Hit, the data is moved from L2 cache to L1 cache.

2.2.3 Branch Prediction Unit

The branch prediction unit uses the algorithm described in section 2.1.3. It maintains a 125 entry history table for the branches.

2.3 Bare Metal Applications on ARM

Bare Metal Applications are programs that are run on the processor without an underlying layer of Operating System. Running a bare metal application is quite different from running an application on top of an Operating System such as Linux.

The job of an Operating System is to manage the resources of the computer system, and provide a mechanism to load and execute programs at the users request. Operating Systems take control of the main memory of the system, and try to make the most efficient use of it by appropriately sharing it among different applications. When an application needs to be executed, the Operating Systems creates space in the main memory, to load the code and the data for the application.

The application code may be loaded at any address in the main memory, at the behest of the Operating System. Hence the code must be compiled in a way such that it is relocatable. The addresses used within the code are relative to the actual (physical) address where the application code will be loaded. During execution, each access to the relative address is translated by the Memory Management Unit, with support from Operating System, to the physical address.

In some sense, it is much more easier to understand how a bare metal application is executed. When a program is compiled to be run as bare-metal application, a specifc address needs to be provided at which the program will be loaded. All memory addresses used in the binary code, will be actual (physical) addresses on the target processor. To run the application, the binary code needs to be loaded at the predefined start address. The processor starts executing the instruction the instruction at this address in a sequential manner, until a branch instruction is found. The processor then jumps to the address provided in the branch instructions and continues executing the code until the end of the program is reached.

It is important to note, that for each instruction and data in the bare metal application, the exact address where it will be located in the memory of the target processor can be known. This information will be used later in the project.

Apart from this, there are some limitations that arise with Bare Metal Applications. Since there is no Operating System, interaction with input and output devices is very difficult. Reading and writing of files from the disk is not possible, because the logic to understand the file system is not available. Using heap memory for dynamic memory allocation, while possible, is complicated. Using Shared libraries in bare-metal applications is not possible, because the Operating System is needed to perform dynamic linking at run-time. All applications must be statically linked.

2.4 Cross-Compiling using GCC

2.5 Compiler Optimizations

The compiler uses a number of techniques to optimize the code, without changing the functionality. Often, this results in a substantially modified control flow in the binary code, compared to the flow in the source code.

Some of these optimization strategies have been discussed here. This will be useful in understanding some of the challenges that the project aims to solve.

2.5.1 Partial or Full Loop Unrolling

To reduce the number of branching instructions that need to be executed, the compiler checks if the number of iterations of a loop can be reduced by merging a few iterations together.

In cases where the number of iterations of the loop are known at compile time, the compiler may try to eliminate the loop by replicating the instructions inside the loop. This may be useful when the number of iterations of the loop is less, and the body of the loop contains only a few instructions. It eliminates the need of branching instructions.

2.5.2 Conditional Execution

Conditional Execution or Predication is a feature that is provided by the Instruction Set for the target processor. Instructions can be predicated with a condition. The instruction is fed to the processor pipeline, but the result of the instruction is only written back (committed) if the condition holds true.

Conditional Execution is used to eliminate conditional branching instructions. A simple *if-then-else* construct is implemented in ARM binary code as a compare instruction, followed by a conditional branch instruction. The condition checks for the result of the compare instruction, and appropriately tells the processor to branch or continue executing the consecutive instruction. Since the result of the compare instruction will not be available until the last stage of the pipeline execution, the next instruction to be executed can not be known with certainty.

The Branch Prediction Unit, predicts whether the branch will be taken or not and accordingly tells the Instruction Fetch Unit to feed the appropriate instruction into the pipeline. If the prediction turns out to be incorrect, the instructions in the pipeline must be flushed and the correct instructions must be reloaded. On the ARM Cortex A5 with an 8-stage pipeline, this will lead to a loss of crucial 5-6 clock cycles.

Conditional Execution can be used to eliminate the use of branching instructions. This will lead to an improvement in performance, especially when the code in *then* and *else* body contained only 1 or 2 instructions. Each instruction in the *then* and *else* body is predicated with an appropriate condition. After the compare instruction, the predicated instructions are fed into the pipeline. The predicated instructions progress in the pipeline, meanwhile the result of the compare instruction is available. The result of the predicated instruction is only written back (committed) if the condition evaluates to true.

If the *then* and *else* bodies contain one instruction each, only one cycles is wasted by using Conditional Execution.

The example in listing [TODO] shows Conditional Execution. Figure [TODO] shows how the control flow of the binary code varies significantly from the control flow of the source code.

2.5.3 Code Rearrangement

The compiler tries to remove data and control dependency among consecutive instructions by rearranging instructions. This leads to reduced pipeline stalls and overall improvement in performance.

2.6 Extracting Debug Information from GDB

GDB is an open source debug tool which can be used to debug applications. It can be used to extract a lot of information. In this project, GDB will be used to extract information about the variables used in the code. This section briefly discusses the scope of information that can be collected from GDB.

For debugging ARM binaries, GDB specially compiled for the target must be used. The following command starts GDB and uses as input *example.elf* which is a binary of a Bare-Metal application compiled to run on an ARM processor. This gives the GDB prompt. GDB can be used for remote debugging on a target hardware, and it also provides a functional simulator for the ARM processor. The following commands are given to tell GDB to use a simulator as a target, and load the binary.

```
$ arm-none-eabi-gdb example.elf
(gdb) target sim
(gdb) load
(gdb) _
```

Often, many variables used in the source code are eliminated (optimized out) during compiler optimization. GDB can be used to extract information about the Global Variables being used in the binary code. It can provide the name, address and sizes of the global variables. To get this information, the following commands can be used.

```
(gdb) info variables
From example.c
unsigned int array[100];
unsigend int result;

(gdb) print &result
$1 = 0x7c8

(gdb) _
```

This shows that global variables *array* and *result* are declared in file *example.c*. It shows the address of the variable *result* using the second command.

Getting similar information about local variables is also possible. The local variables are only visible inside the scope of the function in which they are defined. We need to configure a break point in the beginning of the function, then run the simulation and issue following commands to extract information about local variables defined in the function.

```
(gdb) b main
Breakpoint 1 at 0x368: file example.c, line 10.

(gdb) info locals
i = 0
j = 0

(gdb) print &i
$1 = 0x17b40
```

This shows that local variables *i* and *j* are declared inside the function main. Address of *i* is printed, and it must be noted that this address resides in the stack, as *i* is a local variable.

3 Host Compiled Simulation

Host Compiled Simulation (HCS) is an important area of research in the space of fast simulation techniques for performance estimation. It is popular because it is simple to understand and develop. It is expected to be faster in execution compared to Cycle Accurate Simulators (CAS) and more accurate than other approaches used for Fast Simulation like the Sampling Based Approach.

In this chapter, the concept of HCS is illustrated using a simplified example. The steps involved in performing HCS are outlined. An overview of each step and the challenges is provided.

Detailed implementation of the technique and handling of challenges has been discussed in the next chapter.

3.1 The Technique

HCS is based on the approach of Source Code Instrumentation (SCI). Instrumentation is the technique to modify the source code of an application, so as to extract useful information during the run-time of the application. In HCS, the source code is instrumented to gain information of the time spent in executing the code on a particular Target Processor. The term Target Processor refers to the processor being simulated, and Host Machine is the computer on which the simulation will be run.

When an application is run on a processor, most of the time is spent in following phases.

- Execution of instructions.
- Fetching Data from memory, while the execution is stalled.

The technique is based on the assumption, that number of cycles spent in each phase of the execution can be accurately predicted using instrumentation. The predicted cycles can be accumulated during the run-time of the application to calculate the total cycles spent in execution of the program. The generated data can further be used to estimate the total power spent in the execution.

3.1.1 Simple Example

This simple example will be able to illustrate the concept of Host Compiled Simulation.

Consider the source code in Listing 3.1. The function `sum` calculates the sum of elements in an array and returns the result. The object dump from the binary code generated by the ARM cross-compiler is shown in Listing 3.2. The instrumented code is shown in Listing 3.3, where the annotations are highlighted.

```c
int sum(int array[20])
{
    int i;
    int sum = 0;

    for (i=0; i<20; i++)
        sum += array[i];

    return sum;
}
```

Listing 3.1: Simple C Code

```
00008068 <sum>:
8068:    mov    r3, #0
806c:    mov    r2, r3
8070:    ldr    r1, [r0, r3]
8074:    add    r2, r2, r1
8078:    add    r3, r3, #4
807c:    cmp    r3, #80 ; 0x50
8080:    bne    8070 <sum+0x8>
8084:    mov    r0, r2
8088:    bx     lr
```

Listing 3.2: Objdump Code

```c
unsigned int execCycles;
unsigned int memAccessCycles;

int sum(int array[20])
{
    int i;
    int sum = 0;
    execCycles += 2;
    memAccessCycles += simICache(0x8068, 8);

    for (i=0; i<20; i++)
    {
        sum += array[i];
        memAccessCycles += simDCache(&array + i, READ);
        execCycles += 5;
        memAccessCycles += simICache(0x8070, 40);
    }

    execCycles += 2;
    memAccessCycles += simICache(0x8084, 8);
    return sum;
}
```

Listing 3.3: Instrumented Code

Two global variables **execCycles** and **memAccessCycles** have been declared on lines 1 and 2 respectively. **execCycles** will store the number of cycles spent in actual execution of instructions, when the processor is in active state. **memAccessCycles** will store the number of cycles spent in performing read/write operations to the memory.

From the binary code, 3 basic blocks can be identified. These blocks can be matched to corresponding blocks in the source code. The mapping is shown in the Table 3.1.

Basic Block in Binary		Matching block in Source	
BlockID	Lines	BlockID	Lines
1	1-2	1	3-4
2	4-8	2	7
3	9-10	3	9

Table 3.1: Mapping of Basic Blocks

For each basic block, cycles spent in executing the instructions needs to be annotated. For simplicity, let us assume that the processor has a single stage pipeline, and each instruction takes 1 cycle to execute when input data is available in the registers. The number of cycles spent in execution of each basic block is estimated, and the result is added to the global variable **execCycles** on lines 8, 15 and 19.

The object code contains a load instruction on line 4, which corresponds to loading of the elements of the array. To estimate the time spent in fetching this data, the cache hierarchy of the target system must be simulated. The cache simulator offers an API **simDCache** to simulate data cache access. It takes as parameters the **address** of the data and a **flag** to tell whether it is a read or write operation. The cache simulator returns the number of cycles spent in performing the memory access. This value is accumulated in the global variable **memAccessCycles**, as seen on line 14.

Also, the instruction cache access must be simulated. This is done at the basic block granularity. The cache simulator offers API **simICache** which takes as parameters **address** of the first instructions in the basic block, and **size** of the basic block in bytes. The cache simulator returns the number of cycles spent in fetching the instructions. The value is also accumulated in global variable **memAccessCycles**, as seen on lines 9, 16, and 20.

The values of **execCycles** and **memAccessCycles** are returned at the end of the simulation.

3.2 The Flow

Figure 3.1 shows a flow-chart depicting the stages involved in performing automatic instrumentation. Each stage is explained in further sections.

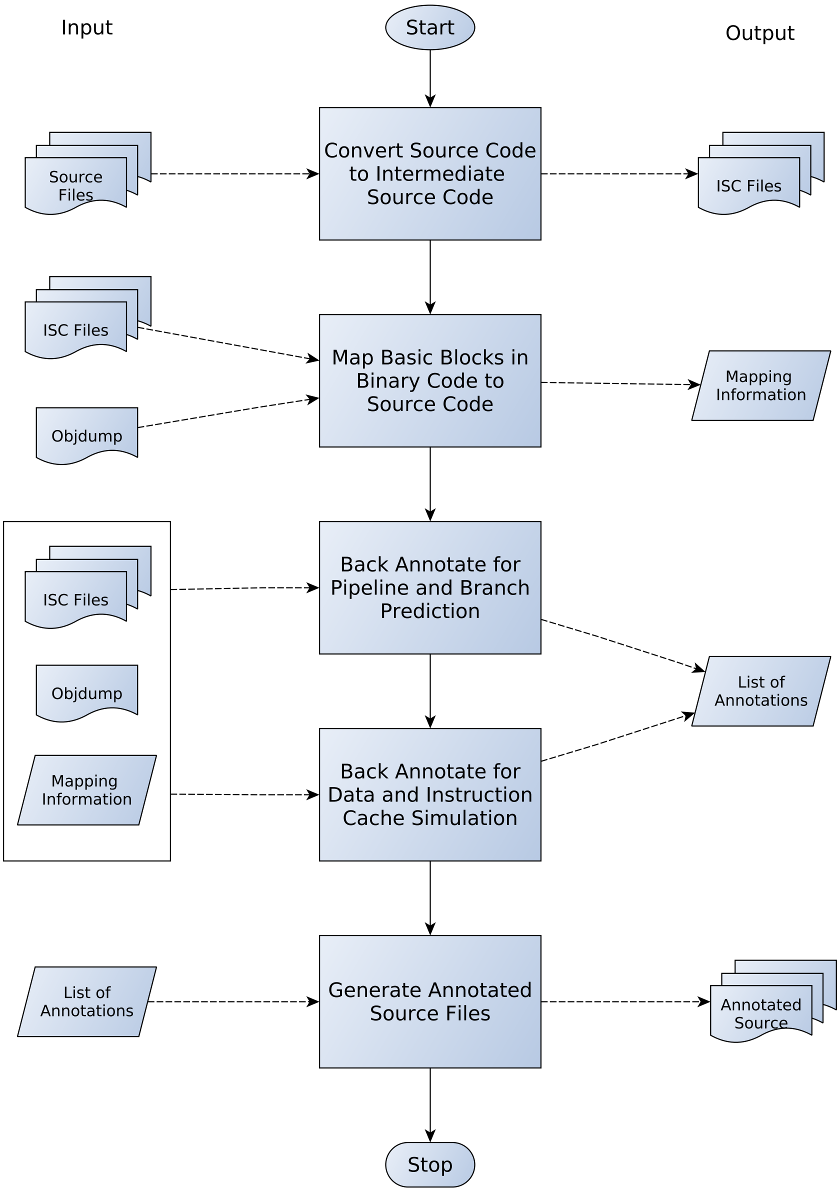

Input
Start
Output
Source Files
Convert Source Code to Intermediate Source Code
ISC Files
ISC Files
Map Basic Blocks in Binary Code to Source Code
Mapping Information
Objdump
ISC Files
Back Annotate for Pipeline and Branch Prediction
Objdump
List of Annotations
Mapping Information
Back Annotate for Data and Instruction Cache Simulation
List of Annotations
Generate Annotated Source Files
Annotated Source
Stop

Figure 3.1: Flow Chart

In this section, a brief overview of the project architecture is provided. Each part of the project is independent, and has been treated as a separate problem. The flow chart in figure [TODO] shows each step of the project. The approach to solve these techniques is presented in the subsequent sections in this chapter. Details of the implementation have been presented in the next chapter.

From the example, it is clear that for correctly instrumenting the code accurate mapping is required between the source code and the binary code. However, this mapping is usually destroyed during the optimization phases of the compiler. The first problem that is solved in this project is to generate accurate mapping information at the Basic Block granularity.

In the next step, GDB is used to derive information about each variable that is used in the application. During compiler optimization, most of the variables defined by the user are optimized out, hence this information can only be extracted from the debug information with the binary code. Names, addresses and sizes of Global and Local Variables are extracted.

For simulating data cache, each load and store instruction is accurately matched to an instruction in the source code. The variable being accessed is identified, and the annotation to simulate the memory access is added to the code. For simulating the instruction cache, annotation is added for each basic block in the binary code to the corresponding basic block in the source code.

To estimate the amount of cycles spent in execution of a basic block in the processor pipeline, each instruction in a basic block is sequentially parsed to identify interlocking. Interlocking occurs when there is a Control or Data Dependency between instructions, which results in pipeline stall for a few cycles. The cycles used by each basic block are annotated back to the code. Further, a mechanism to emulate Branch Prediction Unit has been implemented.

3.3 Source Code to Intermediate Source Code

From the example, it is clear that mapping between source code and binary code is very important for accurate instrumentation. Unfortunately, this mapping is destroyed during the optimization phases of the compiler. The compiler optimizes code by moving instructions around, eliminating branch instructions and modifying loops. Some of these optimization strategies have been discussed in Section [TODO].

The compiler performs these optimizations in two stages. The front-end of the compiler, translates the source code from a high-level language like C, to an Intermediate Representation (IR) called GIMPLE. The processor independent optimization strategies are applied to the IR code. In the back-end of the compiler, the optimized IR code is translated into Machine Language for the target processor. The processor dependent optimization strategies are applied in this phase.

The Intermediate Code is expected to have a control flow similar to the control flow of the Binary Code, since most optimization strategies have already been applied. It should be much easier to perform mapping between the IR Code and the Binary Code. However, instrumentation of IR Code is difficult.

In this project, the source code of the benchmark application is cross-compiled, and the optimized IR Code is translated back into a high-level language, C. The generated code is called Intermediate Source Code (ISC). The generated code is optimized version of the source code, with exactly similar functionality. It is easier to parse and instrument.

This approach is inspired from the research published in [TODO]. The python tool to translate GIMPLE code to C Code has been reused with minor modifications.

3.4 Mapping Algorithm

For accurate instrumentation, mapping at the Basic Block granularity is needed. GDB provides mapping information, but this has been observed to be highly inaccurate. This is a challenging problem and considerable research has been performed to develop algorithms for extracting accurate mapping.

The compilers employ a number of optimization strategies in which the control flow of the source code is modified. One such strategy is briefly discussed below, to illustrate the complexity of the mapping problem. Listing 3.4 shows a simple **if-then-else** construct written in C. The code checks if **a** is greater than **b**. If true, the value of **a** is assigned to **max**, else the value of **b** is assigned to **max**.

Listing 3.5 is representative of how the assembly code may look like without any optimization.

```
int max(int a, int b)
{
    int ret;
    if (a > b)
        max = a;
    else
        max = b;
    return ret;
}
```

Listing 3.4: Example C Code

```
1  00008068 <max>:
2      8068:      cmp    r1, r0
3      806c:      ble    8078
4      8070:      mov    r0, r1
5      8074:      b      807c
6      8078:      mov    r0, r0
7      807c:      bx     lr
```

Listing 3.5: Object Code

Each instruction is loaded in the processor pipeline. While the compare instruction on line 4 is being executed, the branch instruction on the next line is loaded into the pipeline. Depending on the result of the compare instruction, the branch will be taken or not taken. Since the result of the compare instruction is not yet available, the move instruction on line 6 is loaded into pipeline by assuming that the branch will not be taken. However, if the content of register **r1** is lesser or equal to that of **r2**, the pipeline will be flushed and move instruction on line 8 will be executed. The pipeline flush leads to a loss of a few cycles.

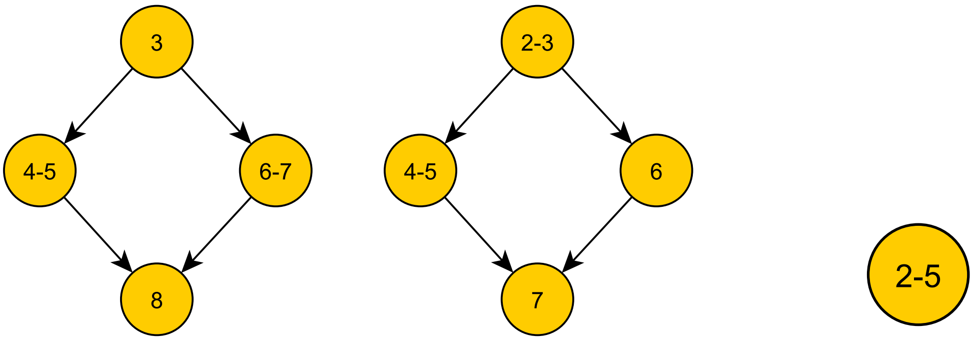
To reduce this loss, the Instruction Sets of most modern processors support a mechanism known as Predication or Conditional Execution. Each instruction like **mov** or arithmetic instructions can be predicated with a condition. The instruction is loaded into the pipeline, but the result is only written back (or committed) if the condition evaluates to true. This feature can be used to eliminate conditional branch instructions. The optimized binary code is as shown in Listing 3.6.

```
00008068 <max>:
    8068:      cmp    r1, r0
    806c:      movge  r0, r1
    8070:      movlt  r0, r0
    8074:      bx     lr
```

Listing 3.6: Optimized Object Code

Since the branching instructions are eliminated, the code in Listing 3.6 will be treated as a single basic block. The control flow graphs for the given source code, and the unoptimized binary code have been represented in figures 3.2a and 3.2b. They are similar, and hence easy to map. Figure 3.2c shows the Control Flow Graph of the optimized binary code. As expected, it is a single block.

Similarly, other optimization strategies may modify the Control Flow. To map the Control Flow Graphs, a standard graph matching algorithm using Depth First Traversal has been implemented, and special handling has been done for each optimization



(a) Source Code

(b) Unoptimized Binary Code

(c) Optimized Binary Code

Figure 3.2: Control Flow Graphs

strategy used by the compiler. Additionally, mapping information from GDB is used in corner cases.

During compile time, developers can tell the compiler which optimizations to perform using the option **-O** followed by a number between 0 and 3 for the level of optimizations, 3 being the highest level of optimization. The tool currently provides promising results at optimization level 1, and can be extended to handle other optimization techniques.

The Control Flow Graphs generated from ISC and the Optimized Binary Code, along with the mapping information generated by the algorithm is stored in a data structure. The data is used in the next steps.

3.5 Annotation for Execution Cycles and Branch Prediction

The annotation for number of cycles spent in execution of the instructions is done at a basic block granularity. Cycles are estimated for each basic block in the binary code, and back annotated to the mapping block in the source code.

To accurately estimate the number of cycles spent in executing the instructions in a basic block, the pipeline architecture of the target processor must be understood. The target processor used for this research is ARM Cortex A5 which has an 8 stage pipeline. As is common with pipelined architectures consecutive instructions may face interlocking, which will lead to pipeline stalls.

Details about the architecture of ARM Cortex A5 are covered in Section [TODO].

3.6 Extracting Debug Information from GDB

As discussed briefly in the example, each load/store instruction in the binary code must be matched to a variable in the source code to simulate Data Cache Access. To be able to do this, information about each variable used in the code must be extracted. GDB has been used to extract this information.

Section **??** describes how GDB can be used to extract names, addresses and sizes of each Global and Local Variable. GDB also offers a mechanism to use scripts, which operate a series of commands and generate an output, which can be later parsed to gather the information required. This mechanism of GDB has been used.

The tool automatically generates GDB scripts and runs them, to extract information about the variables used in the program. For each variable the following information is collected.

- `name` of the variable
- `scope` of the variable. Empty for Global Variables, name of the containing function for local variables.
- `address` of the variable. For Global Variables it is the physical address. For local variables, it is the address relative to the Stack Pointer.
- `type` of the variable.
- `size` of the variable.

3.7 Data Cache Simulation

To estimate total time spent in fetching data from the memory, the tool performs detailed Cache Simulation. Each load/store instruction in the binary code is matched with to an instruction in the source code. The address from which the data is being fetched is identified, and memory access for the address is simulated.

Generally speaking, the host and target processors may use a different memory layout. For example, in some processors memory needs to be allocated in an aligned fashion for better performance, however this may not be the case for another processor. Also, the sizes for the basic data types may differ among different architectures. Size of an integer in one architecture may be 4 Bytes and in another may be 2 Bytes. For accurate cache simulation, the address at which the variable resides in the Host Machine, can not be used. Instead, each memory access as it would occur on the target processor must be simulated.

The challenge lies in extracting the address from which the data is being fetched. This address can not be easily extracted from the cross-compiled binary by static analysis. To extract this address, the tool implements a mechanism to reconstruct each memory access as it occurs on the target processor. The approach is based on the research published in [TODO].

An application uses memory to read and write input and output data. For simplicity, let us consider how an application written in C Programming language uses memory. The memory can be used by the application in 3 ways.

- **Global Memory.** Data stored in Global Memory is accessible by all functions in the program. The memory is stored in a fixed size Data Section. Size of each Global Variable must be known at compile time, so it is called statically allocated memory. In bare metal applications, the physical address of each global variable is known after compilation, and can be extracted from the binary.
- **Local Memory.** Each function can define variables which can only be used inside the function. The content of local variables is stored in the stack frame of the function. The size of the variable must be known at compile time. The addresses used by the local variables at run-time can not be known by static analysis, however the address relative to the Stack Pointer can be extracted.
- **Heap Memory.** Applications can also allocate memory at run-time. The content of this dynamically allocated memory is stored in a special section known as Heap. The heap can grow and contract at run-time. The address and sizes of this type of memory can not be known statically.

The current implementation of the tool, only focuses on simulating access from the Local and Global Memory. Cache Simulation for Heap Memory is complicated, because the memory allocation algorithm used in the target processors have to be emulated to know the exact address where the memory will be allocated. This means, that the tool can only be used with Benchmark applications that do not use Dynamic Memory. This does not limit the importance of this tool since the goal is to simulate for performance analysis, and not functional verification.

The following section explains how each load/store instruction in binary code is mapped to a variable in the source code. Further, the approach to reconstruct the memory access is explained. Implementation of the cache simulator is explained in brief.

3.7.1 Memory Access Reconstruction

The tool parses binary code and emulates each instruction. It maintains the state of registers and updates it, as per the instructions. Branch instructions are ignored. The load/store instructions use register addressing modes to access data. From the content of the register the address of memory being accessed can be known. The memory being accessed may be an array, and this can not be clearly identified at this point.

The addresses of the Global Variables are known. Also, the addresses of Local Variables relative to the Stack Pointer are known. Using this information, the address of the load/store instruction found by emulation can be mapped to one of the variables.

Variables accessed in each basic block of the binary code are recorded, and this information will be used to map the load/store instruction to a specific line in the source code. For each basic block in the binary code, the basic block in the source code is parsed to find the line that causes the memory access. This is needed because the for accessing an array, the index to be added to the base pointer can only be extracted from the source code.

Once this information is known, the memory address for each access can be reconstructed.

3.7.2 Annotation for Data Cache Simulation

Global Variables

For each Global Variable, say "*var*" of any data type, another global variable "*var_addr*" of type "*unsigned long*" is declared in the instrumented code to store the address where "*var*" will be held in the target memory. This address was extracted using the method described in Section 3.6.

The line in the source code where the global variable is being accessed is identified using the technique presented in Section [TODO]. Instrumentation to simulate cache is added after this line. The Cache Simulator is implemented in C language. It offers the following API to simulate Data Cache Access.

```
/**
 * @brief API Function to simulate Data Cache Access
 *
 * @param address The address being accessed.
 * @param isReadAccess Flag to indicate type of access.
 *                     True, if Read Access.
 *                     False, if Write Access.
 *
 * @return number of cycles spent in performing the memory access.
 */
unsigned long long simDCache (unsigned long address,
                              unsigned int isReadAccess)
```

The source code is appropriately instrumented using the above API. For accesses to elements in an array, the index multiplied by the size of the data type is added to the base address. The return value is the number of cycles spent in performing the memory fetch. This value is accumulated in a global variable, in a similar way as shown in the simple example above.

```
int result;
unsigned long result_addr = 0x88ac;            // <--
int input_array[20];
unsigned long input_array_addr = 0x88b0;              // <--

void foo()
{


}
```

Local Variables

The approach for simulating access of Local Variables is quite similar to the approach used for global variables. A new local variable is declared for each local variable used in the function, with a suffix "`_addr`" and data type "`unsigned long`". This variable contains the address of the local variable, relative to the current stack pointer. To accurately estimate the physical address where the memory resides, the value of the Stack Pointer is needed.

The stack grows and contracts during the run-time of an application. Whenever a function is called, a stack frame is created and the stack pointer is incremented. The stack frame contains the values of the function parameters, the local variables used in the function and the return address for the function. The size of the stack frame for each function can be extracted from the binary, and the start address of the stack is fixed at compile time.

To maintain the value of the stack pointer, a global variable "`CSIM_SP`" is added to the source code, which is initialized to the start address of the stack. At the beginning of each function, the value of "`CSIM_SP`" is incremented by the size of the stack frame for the function. The address of the local variable, relative to the stack pointer is added to "`CSIM_SP`", to calculate the physical address of the local variable, as would occur on the target processor.

The memory access is simulated using the same API as above.

Function Parameters

A func

Register Spilling

3.7.3 Implementation of Cache Simulator

3.8 Instruction Cache Simulation

3.9 Annotation for Execution Time in Pipeline

3.10 Annotation for Branch Prediction

4 Implementation

5 Results

6 Conclusion

List of Figures

List of Tables