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

Abstract **iii**

1 Introduction **1**

1.1 Simulation 1

1.2 Related Work 1

1.2.1 Sampling Based Approach 1

1.2.2 Host Compiled Simulation 2

1.3 Focus 2

2 Host Compiled Simulation **3**

2.1 Simple Example 3

2.2 The Flow 6

2.3 Source Code to Intermediate Source Code 7

2.4 Mapping between ISC and Binary Code 8

2.4.1 Handling of Conditional Execution Optimization 9

2.5 Annotation for Execution Cycles 10

2.5.1 Branch Prediction 11

2.6 Annotation for Memory Access 12

2.6.1 Cache Simulator 12

2.7 Instruction Cache Simulation 14

2.8 Data Cache Simulation 14

2.8.1 Memory Access Reconstruction 15

2.8.1.1 Resolve address of each variable 15

2.8.1.2 Analyse binary code for identifying load/store operations on variables 17

2.8.1.3 Parse Source Code 18

2.9 Data Cache Simulation 18

2.9.1 Memory Access Reconstruction 19

2.9.2 Annotation for Data Cache Simulation 20

2.9.2.1 Global Variables 20

2.9.2.2 Local Variables 21

2.9.2.3 Function Parameters 21

2.9.2.4 Register Spilling 21

2.9.3 Implementation of Cache Simulator 21

2.10 Instruction Cache Simulation 21

2.11 Annotation for Execution Time in Pipeline 21

2.12 Annotation for Branch Prediction 21

3 Implementation **22**

4 Results **23**

5 Conclusion **24**

List of Figures **25**

List of Tables **26**

Bibliography **27**

1 Introduction

1.1 Simulation

Simulation is the technique to imitate the operation of a real or abstract system. A model is created which represents the key characteristics and behaviour of the system. This model is then operated to analyse how the system will behave in various situations over time. Simulation has proved useful when working with the real system or developing an abstract system is difficult or expensive.

Simulation is widely used in Hardware Software Co-development. In early stages of development, hardware engineers want to analyse in detail the impact of design decisions on performance of the overall system. Fabrication of hardware at each milestone is time-consuming and expensive. Instead, a model of the processor is created and benchmarking applications are run on these models.

The popular approach used for simulation is called Cycle Accurate Simulation (CAS). In CAS the processor micro-architecture is modelled 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 taken into consideration. For analysing bottlenecks, long-running benchmark applications need to be executed. CAS is not suited for such use-cases due to the slow execution speeds.

In this research, a technique to accelerate performance benchmarking of micro-processors has been explored.

1.2 Related Work

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.

2 Host Compiled Simulation

Host Compiled Simulation (HCS) is based on the approach of Source Code Instrumentation (SCI). Instrumentation is the technique to modify the source code of an application, in order to extract debug or trace information at run-time. In HCS, the source code is instrumented to estimate the time spent in executing the application on a particular target processor.

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

- Execution of instructions.
- Fetching Data from memory.

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 trace generated from the run-time can be used for estimating the power consumption.

The information generated is useful for architects to analyse the impact of design decisions on performance and power consumption of the system. In comparison to popular techniques like Cycle Accurate Simulation (CAS), HCS is easier to develop and fast to execute. This greatly optimizes the design space exploration effort.

In this chapter, the concept of HCS is illustrated using a simplified example. The steps involved in performing HCS are outlined. Further, an overview of each step is provided. Detailed implementation of each step has been discussed in the next chapter.

2.1 Simple Example

Consider the source code in Listing 2.1. The function **sum** calculates the sum of elements in an array and returns the result. The source code is cross-compiled for a target processor[1]. Listing 2.2 shows the object dump of the binary code.

To estimate the time spent in executing this code on the target processor, the code is instrumented. The instrumented code is shown in Listing 2.3, where the annotations are highlighted. The instrumented code is then compiled for and run on the host machine[2]. This process is called Host Compiled Simulation. Let us look at each annotation in the

[1] Target processor is the processor that is being simulated.

[2] Host Machine is the computer used by the developer, where the simulation will be run.

code.

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

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

    return sum;
}
```

Listing 2.1: 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 2.2: Objdump Code

```c
1   unsigned int execCycles;
2   unsigned int memAccessCycles;
3
4   int sum(int array[20])
5   {
6       int i;
7       int sum = 0;
8       execCycles += 2;
9       memAccessCycles += simICache(0x8068, 8);
10
11      for (i=0; i<20; i++)
12      {
13          sum += array[i];
14          memAccessCycles += simDCache(&array + i, READ);
15          execCycles += 5;
16          memAccessCycles += simICache(0x8070, 40);
17      }
18
19      execCycles += 2;
20      memAccessCycles += simICache(0x8084, 8);
21      return sum;
22  }
```

Listing 2.3: Instrumented Code

In the 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.

For accurate instrumentation, mapping at basic block[3] granularity is needed between

[3] Basic Block is a portion of the binary code with only one entry point and one exit point. A basic block

source code and binary code. From the binary code, three basic blocks can be identified. These blocks are mapped to corresponding blocks in the source code by static analysis. The mapping is shown in the Table 2.1.

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 2.1: Mapping of Basic Blocks

Time spent in executing the instructions needs to be accounted. For simplicity, let us assume that the target processor executes each instruction in one cycle, and there is no latency in accessing memory. The number of cycles spent in execution of each basic block can be estimated by static analysis. Each basic block is annotated as seen on lines 8, 15 and 19 to accumulate the total cycles spent in the global variable **execCycles**.

To accurately account for the cycles spent in accessing memory, the cache-hierarchy on the target processor needs to be simulated at run-time. Each load/store operation in the binary code is identified. Annotation is added to the source code to simulate the load/store operation. The cache simulator offers the API **simDCache** to simulate data access. It takes as parameter the **address** of the data and a **flag** to tell whether it is a read or write access. The cycles spent in performing the memory access is returned.

A load operation is identified on line 4 in the object code. This corresponds to loading of elements of **array**. The operation is simulated by the annotation on line 14 in the instrumented code. The return value from the cache simulator is accumulated in the global variable **memAccessCycles**.

The cycles spent in fetching instructions from the memory must also be accounted. 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. Annotation for simulating instruction cache access is seen on lines 9, 16 and 20.

This instrumented code is compiled for and run on the Host Machine. The values of **execCycles** and **memAccessCycles** are delivered at the end.

The following sections will build upon this basic concept. Modern processors are more complicated than the target processor used for this illustration. The features of the modern processors will need to be simulated in sufficient detail to extract accurate estimates of performance.

can not contain any branch instructions.

2.2 The Flow

Figure 2.1 shows a flow-chart depicting the stages involved in performing automatic instrumentation. Each stage is explained in further sections.

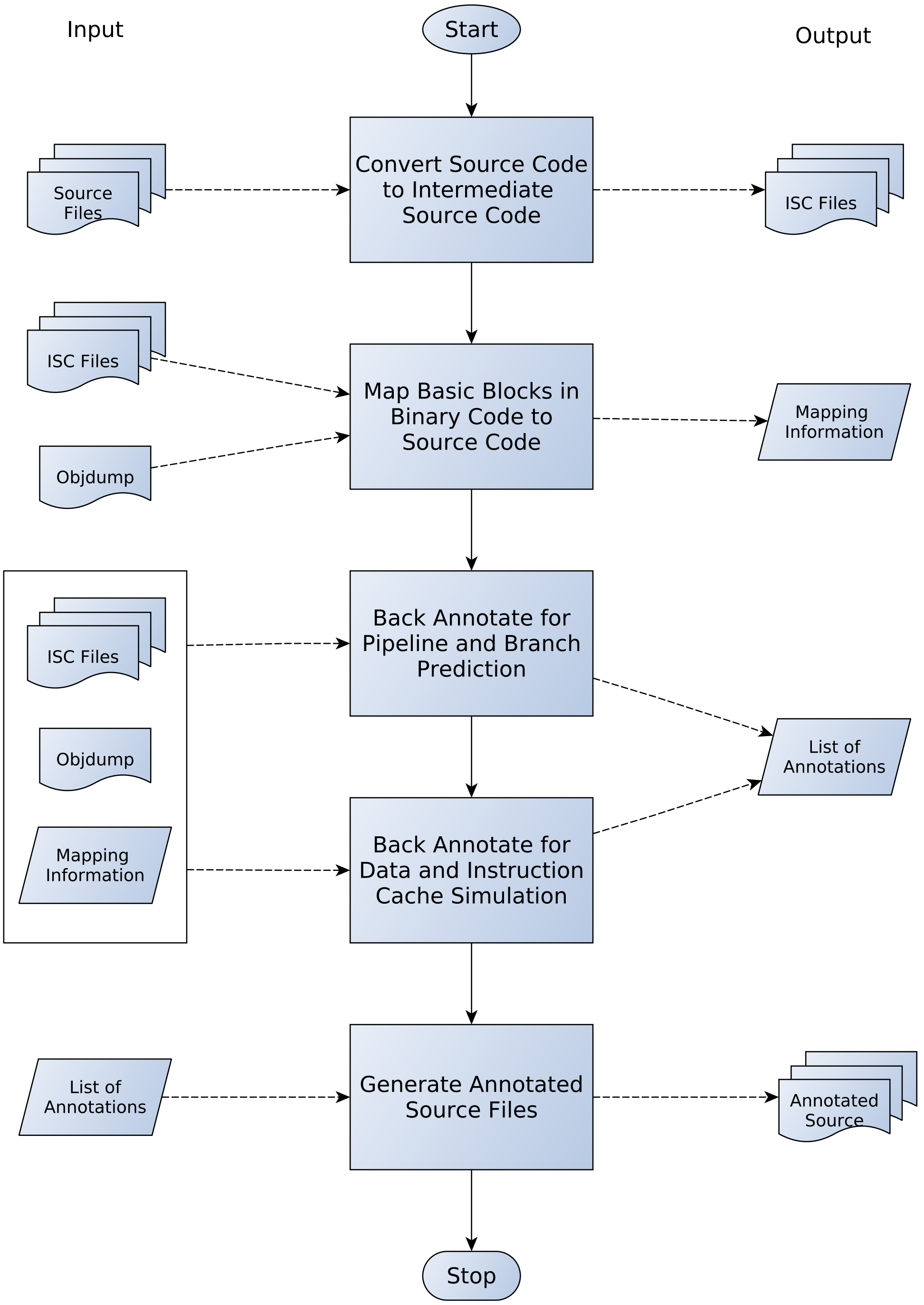

Input
Start
Output
Source Files
Convert Source Code to Intermediate Source Code
ISC Files
ISC Files
Map Basic Blocks in Binary Code to Source Code
Mapping Information
Objdump
ISC Files
Back Annotate for Pipeline and Branch Prediction
Objdump
List of Annotations
Mapping Information
Back Annotate for Data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

Figure 2.1: Flow Chart

2.3 Source Code to Intermediate Source Code

From the example, it is clear that accurate mapping between source code and binary code is very important for instrumentation. Unfortunately, this mapping is destroyed during the optimization phases of the compiler. Extracting accurate mapping information is a challenging problem. In this project, an approach recommended in [TODO] has been used to reduce the complexity of this problem.

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 optimized IR Code has a control flow similar to that of the Binary Code, since front-end optimization strategies have already been applied. It should be comparatively easier to perform mapping between the IR Code and the Binary Code. However, instrumentation of IR Code is difficult as it is in the GIMPLE format.

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 code to convert GIMPLE code to C code has been reused from [1].

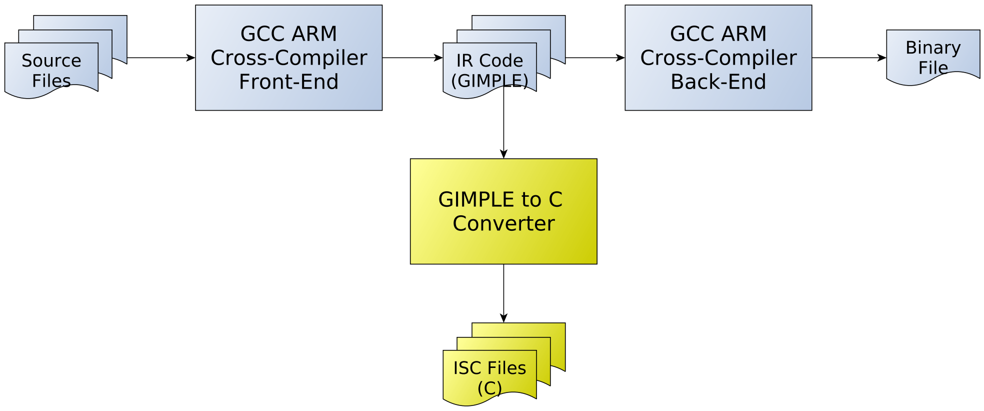


Figure 2.2: Conversion of Source Code to Intermediate Source Code

Extracting mapping between ISC and binary code is comparatively easier. This is aided by the fact that ISC is easier to parse, as it uses simple **if-else** constructs and **goto** commands to implements loops. ISC is fairly easy to read and understand for the developer. Instrumentation is performed to the ISC.

To aid understanding, the ISC will be referred to as the source code in further sections.

2.4 Mapping between ISC and Binary Code

Even between ISC and Binary Code, the control flow is significantly different. This is because processor dependent optimizations are not included in the ISC. Modern processors offer complex optimization features. These features are processor-dependent and can only be utilized by the compiler back-end.

Compiler optimizes code by moving instructions around. Sometimes new blocks will be created in the Binary Code. Using predication or conditional execution, the compiler may try to eliminate branching instructions. Blocks will be merged in doing so. A complete one to one mapping between basic blocks may often not exist. The mapping algorithm must take this into account, and find a mapping such that each basic block in the binary code maps to a basic block in the source code.

GDB provides mapping information, but was observed to be highly inaccurate. To perform accurate mapping, complex techniques have been proposed based on analysis of Data and Control Flow. In this project, the Control Flow Graphs (CFGs)[4] of ISC and binary code are analysed using a mapping algorithm. The approach is inspired from [1].

The mapping algorithm is based on the standard Graph Matching Algorithm using recursive Depth First Traversal. CFGs are extracted by parsing the source code and the binary code. To assist in mapping, a *flow* value is calculated for each node in the graphs. The *flow* value for a node is the sum of the *flows* of the incoming edges to the node, which in turn is equally divided among the outgoing edges. The *flow* value of the root node is 1. Only nodes with equal flow value can be mapped to each other.

The mapping algorithm is implemented as a recursive function, which takes as parameters two nodes from the CFGs of the source code and the binary code. The pseudo code is presented below.

Algorithm 1 CFG Mapping Algorithm

```
function MAPPING(BB_src, BB_bin)
    if flow(BB_src) != flow(BB_bin) then return false
    if (BB_src, BB_bin) $\in$ MatchingDict then return true
    // Check for effects due to Compiler Optimization here
    Succ_src = Succ(BB_src)
    Succ_bin = Succ(BB_bin)
```

It is augmented to take into consideration effects that may occur due to each compiler optimizations.

[4] Control Flow Graph is a graph representing flow of control among basic blocks in the code. The nodes represent basic blocks, and the edges represent the possible flow of execution.

2.4.1 Handling of Conditional Execution Optimization

Conditional Execution or Branch Predication is a feature supported in some Instruction Set Architectures to mitigate the cost associated with conditional branching. Consider the following example to understand the performance impact of this feature.

Example 2.1 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**. Example 2.2 is representative of how the assembly code may look like without any optimization.

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

Example 2.1: 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
```

Example 2.2: Unoptimized Object Code

Instructions take more than one cycle to execute. To improve throughput, execution unit in almost all processors is implemented as a multi-stage pipeline. Instructions are fed into the pipeline and executed in parallel.

The **cmp** instruction on line 2 in Listing 2.2 is fetched by the first stage of the pipeline. While it is being decoded in the next stage of the pipeline, the branch instruction on line 3 has been fetched. Depending on the result of the compare instruction, the branch will be taken or not taken. The result of the compare instruction has not yet been evaluated. The processor cannot know with certainty which instruction to fetch after the branch instruction.

By assuming that the branch will not be taken, the processor fetches the **mov** instruction on line 4. However, if the prediction is incorrect the pipeline must be flushed and **mov** instruction on line 6 must be executed next. The pipeline flush leads to loss of multiple clock cycles.

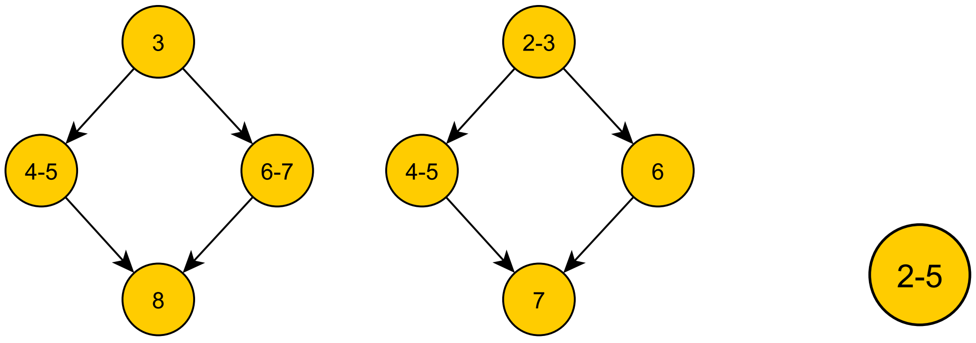
This loss can be reduced by using Conditional Execution, when supported by the architecture. Each instruction can be predicated with a condition. The instruction is executed in the pipeline, but the result is only written back (or committed) if the condition evaluates to true. A few clock cycles can be saved if the length of the conditionally executed block is small. The optimized binary code is as shown in Example 2.3.

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

Example 2.3: Optimized Object Code

Since the branching instructions are eliminated, the code in Example 2.3 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 2.3a and 2.3b. They are similar, and hence easy to map. Figure 2.3c shows the Control Flow Graph of the optimized binary code. As expected, it is a single block.

In such case, all four blocks in the source code should be mapped to the block in the optimized binary code. Occurrence of Conditional Execution instructions in a basic block is identified, and marked in the Control Flow Graph (CFG)



(a) Source Code

(b) Unoptimized Binary Code

(c) Optimized Binary Code

Figure 2.3: Control Flow Graphs

2.5 Annotation for Execution Cycles

The annotation for number of cycles spent in execution of the instructions is done at a basic block granularity. A global variable **execCycles** is declared. Cycles spent in executing each basic block in the binary code is estimated and annotated in the mapped basic block in source code. On entering the basic block, the annotated cycles for the block are added to **execCycles**.

Most processors use a multi-stage pipelined execution unit. The cycles spent in executing instructions depends on the structure of the pipeline. Some instructions have data and control dependency on the previous instruction. For instance, an instruction may need as input the result generated by the previous instruction. This is known as interlocking and leads to pipeline stalls. To accurately estimate the number of cycles spent in executing the basic block the structure of the pipeline and effects of interlocking must be taken into consideration.

Instructions from the binary are parsed sequentially and simulated on the pipeline in a cycle accurate fashion. For each basic block, it is assumed that the pipeline is initially empty, which may not be the case and will be accounted for later. It is also assumed that all the data needed by the instructions is available with no latency. The cycles spent in fetching the data from memory will also be accounted later. Interlocking between instructions is identified and appropriately accounted.

The total number of cycles as calculated from the simulation is annotated to the corresponding basic block in the source code, as illustrated in the example below. On entering the basic block, global variable **execCycles** is incremented by the number of cycles spent in executing the basic block. This is done each time the basic block gets executed.

2.5.1 Branch Prediction

To calculate the cycles spent in execution, it was assumed that the pipeline is empty at the start of each basic block. This may not always be the case. This will be taken into account here.

Conditional branching is implemented in binary code as a compare instruction followed by a conditional branch instruction. The branch is taken depending on the result of the compare instruction. In a pipelined execution unit, instructions are executed in parallel. The result of the compare instruction may not be available in time, to decide the instruction to be fetched after the branch instruction. The processor uses Branch Prediction Unit (BPU) to predict the outcome of the branch instruction. The appropriate instructions are loaded into the pipeline.

If the prediction is incorrect, the pipeline must be flushed and the correct instruction to be executed next must be fetched. If the prediction is correct, a few cycles are saved. To account for the saved cycles, the BPU is simulated.

The BPU, uses heuristics to predict the outcome of a branch instruction. A state machine is implemented to predict the outcome of the branch. A table maintains the history of branch instructions seen in the recent past. The address of the branch instruction is stored along with a 2-bit state information. The states and transitions for each branch are shown in the state machine diagram in figure 2.4.

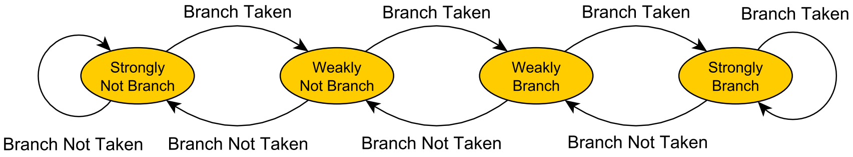


Figure 2.4: State Machine Diagram implemented in the Branch Prediction Unit

The BPU predicts that the branch will not be taken, when the current state is either "Strongly Not Branch" or "Weakly Not Branch". In the other states, the BPU predicts that the branch will be taken.

When a branch instruction is loaded into the pipeline, the BPU checks if the information for the branch is present in the history table. If an entry is not found, the processors predicts that the branch will not be taken. An entry is added to the table with state "Strongly Not Branch". When the branch is found in the history table, the prediction is made based upon the current state of the entry. The state of the entry is changed accordingly, when the outcome of the branch is known.

A Branch Prediction Simulator has been developed which uses the same algorithm. It offers the API as shown in Listing [TODO]. For each basic block in the binary code, the mapped basic block in source code is instrumented to call this API, as illustrated in example in listing [TODO].

The start and end address of the basic block is parsed as a parameter. This information is extracted from the binary code. When the function is called, the simulator checks whether the branch from the previous block to current block would have been predicted by the BPU. It returns **True** if the branch would have been predicted correctly, in which case a few cycles would be subtracted from **execCycles**.

2.6 Annotation for Memory Access

The cycles spent in fetching data and instructions from the memory must be accurately estimated. To do this, a Cache Simulator has been implemented.

2.6.1 Cache Simulator

Most processors use a hierarchy of low-latency cache memories to improve performance by reducing the time spent in fetching data from memory. Caching has a significant impact on performance. To take this into account, an accurate simulation of the cache hierarchy needs to be done.

The Cache Simulator emulates the caching hierarchy of the target processor. It keeps a track of the data stored in the caches. Whenever a memory access is performed, the simulator checks whether the data can be fetched from the caches. If the data is not found in any cache, it must be fetched from the memory. The simulator returns the number of cycles that would be spent in performing the memory access on the target processor.

Most processors use separate cache for Data and Instructions. The cache simulator offers the following API for the different types of memory accesses.

```
/**
 * @brief Function to simulate Data Cache Access.
 *
 * @param Address of the data to be fetched
 * @param True, if read access
 *        False, if write access
 * @param Detailed result for trace
 *
 * @return Number of cycles spent in performing access.
 */
unsigned long long simDCache(unsigned long address,
                             unsigned int is_read_access,
                             struct csim_result_t *csim_result);

/**
 * @brief Function to simulate Instruction Cache Access.
 *
 * @param Start Address of the basic block
 * @param Size of the basic block in Bytes
 * @param Detailed result for trace
 *
 * @return Number of cycles spent in performing access.
 */
unsigned long long simICache(unsigned long address,
                             unsigned long size,
                             struct csim_result_t *csim_result);
```

For each data access in the application, annotation is done in the source code to call function **simDCache** and trigger Data Cache simulation. The address of the data being fetched is provided as parameter, along with a flag to signify whether it is a read or write access. Simulation of Instruction Cache is done at basic block granularity. **simICache** is called, with the start address of the basic block and the size of the basic block in bytes as parameters. Number of cycles spent in performing the memory access is returned by the cache simulator.

The third parameter to both **simDCache** and **simICache** is an address to an object of type **struct csim_result_t** where detailed result of the simulation is stored. This data will be useful for estimating power consumption, as will be discussed later.

Caches may have varying parameters and features such as,

- Sizes

- Approach for Associativity of Data (Direct Mapped, N-way Set Associative)
- Hierarchy (multiple levels of caching)
- Replacement Policies

The cache simulator has been designed in a modular fashion. Platform specific implementation of the cache may be plugged into the API. Architects can alter specific parameters of the cache and analyse the impact on performance, without having to instrument the code again.

2.7 Instruction Cache Simulation

For estimating cycles spent in fetching the instructions from memory, instrumentation for instruction cache simulation is performed at the basic block granularity.

For each basic block in the cross-compiled binary code, the address of the first instruction in the block and size of the block in bytes is extracted. Note that since the binary is compiled to be run on Bare-Metal, the load address of the binary was decided at compile time. The addresses of instructions extracted from the binary, were physical addresses where the instructions will reside in the memory of the target system.

The corresponding basic block in the source code is identified from the mapping information. Instrumentation is performed at the beginning of the basic block, and the instruction access is simulated using the API provided by the cache simulator.

As illustrated in listing [TODO], the return value from **simICache** is accumulated to the global variable **memAccessCycles**.

2.8 Data Cache Simulation

The example shown in Section [TODO] was simplified for illustration. It had a flaw which will result in inaccuracy of estimation. To simulate load operation to fetch elements of **array**, the address of **array** at run-time from the Host Machine was used. For accurate estimation, data access must be simulated using target addresses. This is important since the host and target memory systems may vary significantly. For instance, the host and target system may use different sizes for basic data types, like integers. This will lead to severe inaccuracy when accessing a sequence of elements from an array of integers.

Resolving the address of load/store operation cannot be done by static analysis of the code. The process to do this is inspired from research published in [TODO] and is called Memory Access Reconstruction.

2.8.1 Memory Access Reconstruction

Memory Access Reconstruction is the technique to resolve address of each load/store instruction as it would occur on the target processor. The address is then used for accurately simulating data cache access.

The steps needed for Memory Access Reconstruction are as follows

- Resolve address of each variable used in the program.
- Analyse the binary code to identify the variable being accessed by each load/store instruction.
- Parse the source code to extract information for accurate instrumentation of Data Access

2.8.1.1 Resolve address of each variable

An application may use different types of variables. The technique to resolve address of each variable is described below.

Global Variables Global Variables are accessible by all functions, and are stored in the Data Section of the application memory. The physical addresses of the global variables are decided at compile time. The address, size and type of each Global Variable is extracted by static analysis of the binary using GDB.

For each global variable **var**, another global variable **var_addr** is declared by instrumentation. The address of the global variable on the target system is stored in this variable. This address is later used for simulating access to the global variable. Refer to the example 2.4.

```
int globalVar[20];
unsigned long globalVar_addr = 0x7c8; // Address of global variable

void foo ()
{
    ...
}
```

Example 2.4: Instrumentation to resolve address of Global Variables on Target Device

Local Variables Local Variables are defined inside a function definition, and are only accessible inside the function. Memory for Local Variables is only allocated when the function is called, and is located in the stack frame of the function. The actual physical address of the local variables can not be resolved by static analysis, as the stack grows and compacts at run-time. However, the address of each local variable relative to the value of the stack pointer, can be known.

If the value of the stack pointer is known, the relative address can be added to it to resolve the physical address of the local variable. Whenever a function is called, the stack frame of the function is pushed to the stack. The stack frame is popped when the function returns. To keep track of the stack pointer at run-time, a global variable **CSIM_SP** is declared and initialized with the initial value of the stack pointer. The stack frame size of each function is extracted by static analysis of the binary. **CSIM_SP** is incremented at the beginning of each function by the stack frame size of the function and decremented before the function returns.

Example 2.5 illustrates the annotations needed to resolve the physical address of local variables.

```
unsigned long CSIM_SP = 0x1ff280; // Intial Value of Stack Pointer

void foo ()
{
    double localVar;
    unsigned long localVar_addr = 0x08; // Address relative to SP

    CSIM_SP = CSIM_SP + 0x16; // Increment by size of stack frame

    ...

    CSIM_SP = CSIM_SP - 0x16; // Decrement by size of stack frame

    return;
}
```

Example 2.5: Instrumentation to resolve address of Local Variables on Target Device

Dynamically Allocated Memory Dynamically Allocated Memory is stored in the Heap Section. To allocate and free heap memory, the application uses API provided by system libraries. The memory allocation algorithm of the target system needs to be emulated at run-time of simulation, to resolve physical addresses of dynamically allocated memory. This approach is discussed in [TODO]. However, this is complicated to achieve, and has been ignored for this project. Only benchmark applications that do not use Dynamically Allocated Memory can be used. The project can later be extended to include this functionality.

2.8.1.2 Analyse binary code for identifying load/store operations on variables

To identify which variable is being accessed, the binary code is partially simulated. A simple simulator is developed in Python, which maintains the state of each register in the target processor. Starting from the main function, each instruction in the binary code is parsed and state of the registers is updated. In this simulation, the branching instructions are ignored. This means, each instruction will only be parsed once.

When function calls are identified, the current state of the registers is stored. This state will be used when simulating the called function. This is done to keep track of function parameters that are passed as values in registers. For parameters being passed in the stack, this approach does not correctly work. A queue is maintained for each function that is called. Each function is simulated with the initial state of registers that was recorded when the function was first called. Note that each function will only be simulated once, and subsequent calls to the function were ignored.

By doing this type of partial simulation, the address of each load/store instruction can be extracted. This information helps in identifying which variable is being accessed by the load/store instruction.

- If the address belongs to Data Section, the instruction must be accessing a Global or Static Variable. The address is compared with the address of each Global Variable extracted from GDB in previous phase. The correct variable being accessed is identified.
- If the address belongs to the Stack Space, it could be accessing a local variable. The variable being accessed is again identified from the information collected in the previous phase. Additionally, the stack space is used to spill registers. The load/store instruction could be associated with this operation. This can be distinctively identified, if no local variable is located on this address.

The memory accesses performed in each basic block in the binary code are recorded. Instrumentation for simulating these accesses will be done in the corresponding basic blocks in the source code.

This approach has certain limitations. It may fail, in the presence of some pointer dereferencing operations. When the variable being accessed by a load/store instruction is not identified, appropriate hints are provided in the log to enable the user of the tool to manually instrument the code.

Note, that in this phase only the variables being accessed were identified. The variable may be an array of a stream of data which was been accessed with an index in a loop. Since branching instructions were ignored, this information could not be extracted. For extracting this information, the source code needs to be parsed.

2.8.1.3 Parse Source Code

By analysing the binary code, variables accessed in each basic block were identified. Additionally, load/store instructions associated with register spilling were distinctly identified. Instrumentation for register spilling is straight forward.

In this stage, the focus is to identify the statement in source code which causes memory access to a variable. By parsing this statement, the exact address being accessed can be identified. A custom C Parser is implemented in Python to parse the Instrumented Source Code (ISC). Conversion of source code to ISC helps here, because ISC is easier to parse. The parser returns a list of variables being accessed by the statement, along with other information like index expression and whether it is a read or write operation.

Matching statement is identified for each memory access in the binary code. In some cases, matching statement will not be found.

2.9 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.

2.9.1 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.

2.9.2 Annotation for Data Cache Simulation

2.9.2.1 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 **??**.

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()
{


}
```

2.9.2.2 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.

2.9.2.3 Function Parameters

A func

2.9.2.4 Register Spilling

2.9.3 Implementation of Cache Simulator

2.10 Instruction Cache Simulation

2.11 Annotation for Execution Time in Pipeline

2.12 Annotation for Branch Prediction

3 Implementation

4 Results

5 Conclusion

List of Figures

2.1 Flow Chart 6
2.2 Conversion of Source Code to Intermediate Source Code 7
2.3 Control Flow Graphs 10
2.4 State Machine Diagram implemented in the Branch Prediction Unit . . 12

List of Tables

2.1 Mapping of Basic Blocks 5

Bibliography

[1] S. Chakravarty, Zhuoran Zhao, and A. Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.