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
Advisor:	Mr. Bo Wang
Advisor:	Dr. Josef Weidendorfer
Supervisor:	Prof. Michael Gerndt

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 or Virtual Prototyping is a widely used technique for Hardware/Software Co-development. A model of the target processor[1] is created which represents the key characteristics and behaviour. It is used by hardware architects to analyse the performance impacts of design decisions at early stage in Embedded System development. Software Engineers use it as a tangible environment for early software development, and to analyse and optimize software performance. Simulation helps reduce critical issues, which would otherwise become visible only in the Integration Phase.

The popular approach is called Instruction Set Simulator (ISS). Each instruction in the binary code is read, and the model is operated according to the instruction. For highly accurate timing estimation, the micro-architecture of the processor is modelled in great detail. This is called Cycle Accurate Simulation (CAS). With ever increasing complexity of processors, developing and maintaining CAS is tedious and time-consuming. CAS is not suited for simulating long-running software scenarios due to the inhibiting slow speeds of execution. ISS can be performed at higher levels of abstraction to increase the execution speed at the cost of accuracy of estimation. However, since each instruction must be read and executed, the speed of execution is much slower compared to native execution speed of the target processor.

Researchers have focussed on developing better techniques for benchmarking embedded systems. The focus is on reducing complexity, improving execution speed and providing sufficiently accurate estimates of performance. Such a technique will help architects in identifying critical issues earlier and leaving more time for engineers to iterate and explore the design space.

Host Compiled Simulation (HCS) is a popular reserach topic in this area. HCS is based on the technique of instrumentation. Source code of an application is analysed and instrumented to extract trace information which will be used to estimate the time spent in execution on the target processor. The instrumented code is run on the Host Machine[2].

In this work, the technique to automate the instrumentation of source code has been developed. The approach has been extended to provide estimates of power consumption. This approach can speed up iterations in the Design Space Exploration Phase, and reduce the cost and effort. Results from various benchmark suites show 98% and higher accuracy of estimates with average speeds of execution of 2000 MIPS.

[1]The term target processor is used to refer to the processor that is being simulated.
[2]The term host machine is used to refer to the computer where the simulation will be run.

Contents

1. Introduction

1.1. Simulation

Simulation is the technique to imitate the operation of a real-world system. A model or a virtual prototype is developed which represents the key characteristics and behaviour of the system. Simulation is the process of operating this model to analyse how the system will behave in the various situations. Simulation is frequently used when working with the real system is difficult or impossible.

Simulation is widely used in Hardware Software Co-development. In early stages of processor development, architects want to analyse the impact of minor design decisions on the overall system. The balance between performance and power consumption is critical and delicate. Fabrication of prototypes to measure the impact of each modification, is an expensive and time-consuming process. Instead, a Virtual Prototype of the processor is created and simulated for analysis.

Virtual Prototyping is a very popular approach in the industry today. This greatly reduces the cost and effort in the Design Space Exploration Phase. By saving time, engineers can now iterate on design decisions more quickly. It also allows architects to identify critical issues at an early stage. These issues would have otherwise emerged during Integration Phase. In addition, the Virtual Prototypes serve as a tangible environment for early software development.

1.2. Popular Techniques

1.2.1. Cycle Accurate Simuation (CAS)

In this technique, the micro-architecture of the processor is modelled in great detail. Each stage of the execution pipeline is modelled along with other building blocks of the processor like Cache and Branch Prediction Unit. This approach provides a cycle accurate estimation 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 software scenarios are executed. CAS is not suited for such use-cases due to the slow execution speeds.

1.2.2. Functional Simulation

In this technique, simulation is carried out at a higher level of abstraction. The state of registers in the target processor is maintained by the simulator. Each instruction of the cross-compiled binary code is read and state of the registers is updated. This is known as Functional Simulation.

This technique is used for functional verification, and as an environment for early software development. Since the details of the processor are ignored, performance estimates can not be extracted.

1.3. Motivation

With ever increasing complexity of processors, developing CAS is very difficult. The inhibiting slow speed of execution is another impediment to the relevance of this technique. Researchers have focussed on developing techniques to accelerate performance benchmarking of micro-processors.

Host Compiled Simulation or Source-Level Simulation is a popular research area [TODO: References]. HCS is based on the approach of Instrumentation. Source code of a benchmarking application is modified to extract important trace information. This trace information will be used at run-time to analyse how the application will perform on a target processor. Additionally, this trace information can be used to estimate the power consumed by the target processor in running the benchmark application. The instrumented source code is compiled and run on a Host Machine.

In comparison to CAS, HCS is easy to understand and develop. It can provide a very high accuracy of estimation, while running at much higher speeds.

In this thesis, the approach of HCS has been explored. A tool to automate instrumentation of source code has been developed. Estimation of execution time using this approach has been covered earlier[TODO: References]. The contribution of this research is the technique to estimate the power consumption.

1.4. Related Work

1.4.1. 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 faster Functional Simulation, and some samples are executed using the detailed CAS. The number of cycles spent in execution of the samples is calculated. The results are interpolated to estimate the number of cycles spent in executing the entire application.

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.

1.5. Thesis Outline

Chapter 2 presents the concept of Host Compiled Simulation from a general perspective. The overall flow of the technique is discussed in detail, along with the challenges at each step.

Chapter 3 describes the features of the target processor that have been implemented in the simulation.

Chapter 4 presents a detailed analysis of the results.

Chapter 5 provides conclusion and pointers for future contribution.

2. Host Compiled Simulation

Host Compiled Simulation (HCS) is a popular research topic as a techniques to accelerate the benchmarking of processors. 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. The instrumented source code is run on the host machine, and hence the name.

While earlier research has mostly focussed on using HCS for timing estimation, in this work the technique has been extended to estimate the power consumption.

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 also be used for estimating the power consumption.

In comparison to Cycle Accurate Simulation (CAS), HCS is supposed to be much faster to execute since details of the processor micro-architecture are not simulated at run-time. The technique is also easier to understand and implement. From earlier research, it has shown a promise of highly accurate estimation [TODO: References].

This technique could prove to be a competitive edge in the industry, as it will allow architects to better analyse the performance of the system at an early stage in development. Engineers will be able to iterate faster, and explore the design space extensively.

In this chapter, the concept of HCS is illustrated using a simplified example. Then, each step involved in performing HCS is described in detail. References to appendix have been made for implementation specific details where needed.

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 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
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 2.3: Instrumented Code

In the instrumented code, two global variables **execCycles** and **memAccessCycles** have

[1]The term target processor is used to refer to the processor that is being simulated.
[2]The term host machine is used to refer to the computer where the simulation will be run.

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 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.

[3]Basic Block is a portion of the binary code with only one entry point and one exit point. A basic block can not contain any branch instructions.

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.

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.

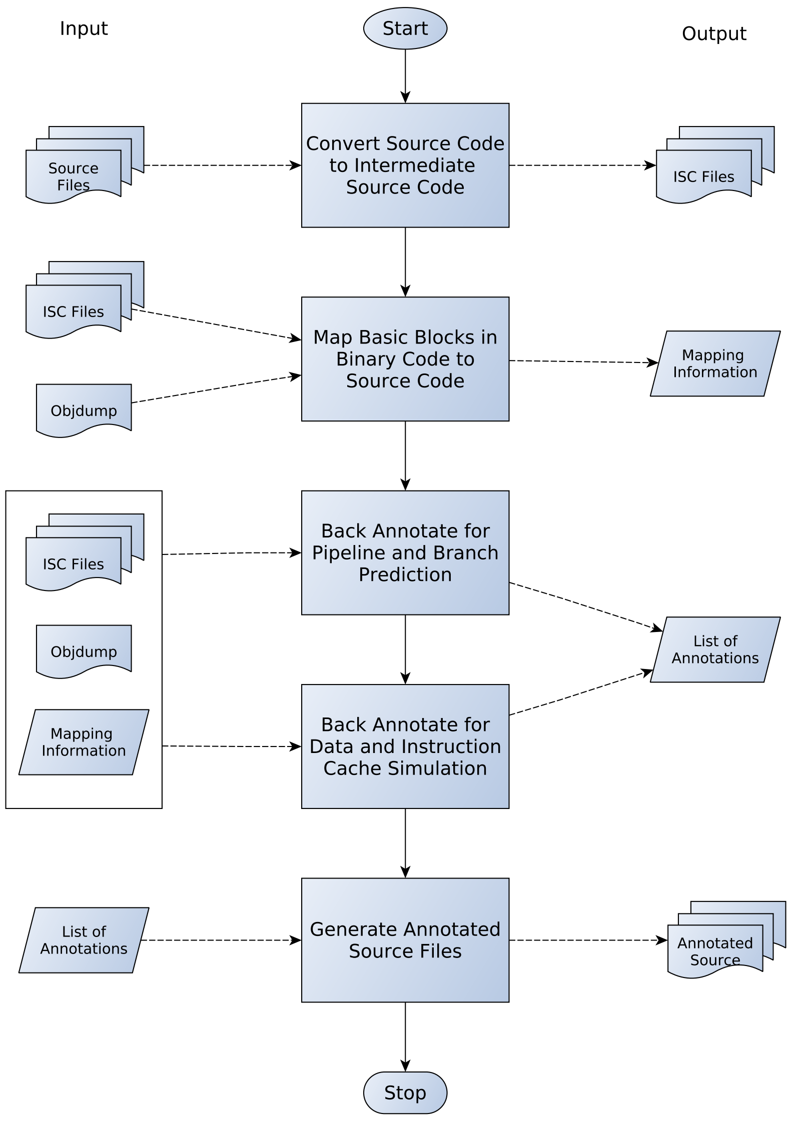

Input
Start
Output
Source Files
Convert Source Code to Intermediate Source Code
ISC Files
ISC Files
Map Basic Blocks in Binary Code to Source Code
Mapping Information
Objdump
ISC Files
Back Annotate for Pipeline and Branch Prediction
Objdump
List of Annotations
Mapping Information
Back Annotate for Data 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 [3] 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 in the front-end. 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 [3].

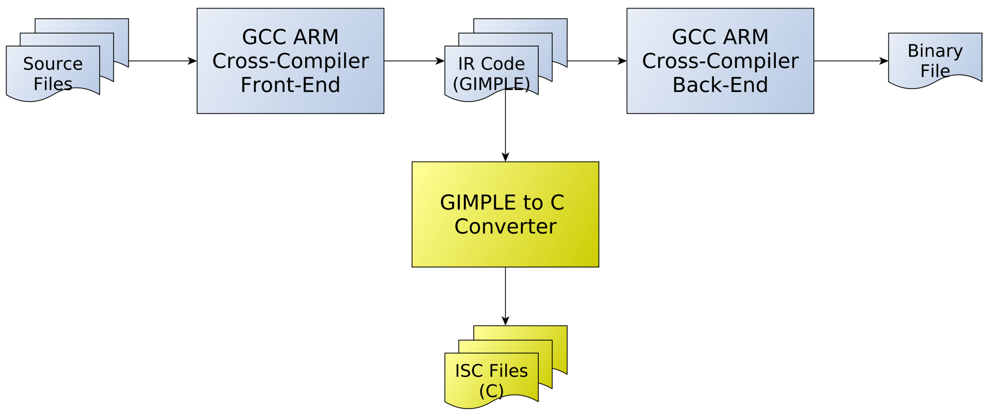


Figure 2.2.: Conversion of Source Code to Intermediate Source Code

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 interchangeably referred to as the source code in further sections.

A sample of ISC code generated by the tool is shown in Appendix A.

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. New basic blocks may be created and some blocks may get merged in the binary code. 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 one or many basic blocks in the source code.

GDB can be used to extract mapping between instructions in binary code and statements in the source code. However, this mapping 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 [3]. The mapping algorithm is based on the standard Graph Matching Algorithm using recursive Depth First Traversal.

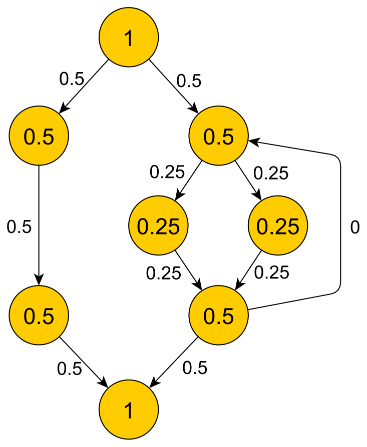


Figure 2.3.: Computation of Flow Value

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 to successor nodes. *Flow* for backward edges is 0. The *flow* value of the root node is 1. Only nodes with equal flow value can be mapped to each other. Figure 2.3 shows the computation of flow value in a graph. The labels on the nodes and edges represent the flow value.

The pseudo code of the **mapping** function is presented in Algorithm 1. The recursive

[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.

function takes two nodes as parameter, and tries to find if the nodes map to each other. It checks that the nodes have the same *flow* values, and returns *False* if not. If the nodes have already been mapped to each other, the function returns *True*.

Algorithm 1 CFG Mapping Algorithm

```
function MAPPING(BB_src, BB_bin)
    if flow(BB_src) != flow(BB_bin) then
        return false
    end if

    if (BB_src, BB_bin) ∈ MappingDict then
        return true
    end if

    // Check for effects due to Compiler Optimization here

    SS_src = Succ(BB_src)
    SS_bin = Succ(BB_bin)
    if len(SS_src) != len(SS_bin) then
        return False
    end if

    for S_src in SS_src do
        for S_bin in SS_bin do
            if Mapping(S_src, S_bin) == True then
                break
            else
                continue
            end if
        end for
        // Mapping could not be found for S_src
        return false
    end for

    // All children mapped; Input Nodes must map to each other
    MappingDict ← (BB_src, BB_bin)
    return true

end function
```

At this point, special handling for different compiler optimizations is done. Section 2.4.1 illustrates the handling for Conditional Execution optimization that is frequently seen in the test set of benchmarks.

Hereafter, the successor sets of both nodes are created. Two nodes can be mapped to each other, only if they have the same number of successor nodes. For each successor **S_src** in **SS_src**, a matching node in **SS_bin** is identified by recursively calling the

mapping function.

The nodes map to each other, if a mapping can be found for all successor nodes. The algorithm tries to find matching path from the start node, to the exit node and creates mapping between the nodes on this path.

The mapping algorithm has limitations. For instance, it can not differentiate between the **then** and the **else** block in an **if-then-else** construct. Mapping from GDB is used to resolve such situations.

The mapping algorithm might fail in some corner cases. For the benchmark applications used for testing, the algorithm was found to work accurately. The instrumentation tool generates graphical representation of the mapping between the CFGs. Users can easily verify the accuracy of mapping by analysing this output.

Mappings generated for some benchmark applications that were used for testing have been shown in Appendix B.

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

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 Example 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.

```
1 00008068 <max>:
2     8068:      cmp    r1, r0
3     806c:      blt    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

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. 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. In the optimized code in Example 2.3 the **movge** instruction will be executed, but the value of **r1** will be written to **r0** only if the result of the compare instruction is "Greater than" or "Equal". A few clock cycles can be saved if the length of the conditionally executed block is small.

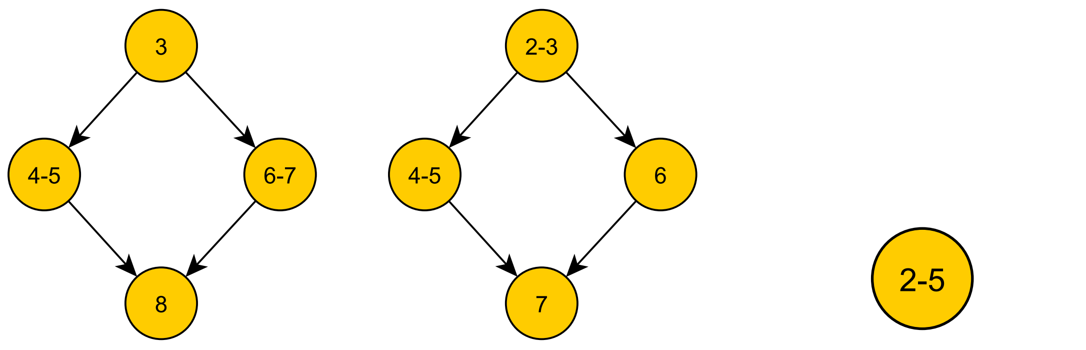
```
00008068 <max>:
    8068:      cmp    r1, r0
    806c:      movge  r0, r1
    8070:      movlt  r0, r0
    8074:      bx     lr
```

Example 2.3: Optimized Object Code

The CFGs for the source code, and the unoptimized binary code have been represented in figures 2.4a and 2.4b. The labels in the nodes represent line numbers for the corresponding basic blocks. These graphs are similar, and hence easy to map. However, in the optimized binary code the branching instructions are eliminated. The code is considered as a single basic block, as shown in the CFG in figure 2.4c.

To handle this optimization, the binary basic block is analysed to check if Conditional Execution Instructions have been used. The corresponding basic block in source code must have a conditional branch, which can be identified by analysing the CFG. If the length of each conditional execution block in source code is smaller than a threshold, the algorithm predicts that the blocks have been merged using Conditional Execution. All four blocks in the source code are mapped to the single block in the binary code.

The mapping continues between the source code basic block after the **if-then-else**



(a) Source Code (b) Unoptimized Binary Code (c) Optimized Binary Code

Figure 2.4.: Control Flow Graphs

construct, and the current binary basic block. If mapping is not found by this approach, the algorithm will recursively visit this step, and try another approach.

2.5. Annotation for Execution Cycles

The estimation of number of cycles spent in execution of the instructions is done at a basic block granularity. Cycles spent in executing each basic block in binary code is estimated by static analysis. The cycles are annotated in the mapped basic block in the source code. Each time when the basic block is executed, the annotated cycles are accumulated in a global variable.

In the example in Section 2.1, a very simple processor was used where each instruction takes one cycle to execute. Estimation of cycles consumed in executing a set of instructions in such a processor is easy. In reality, each instruction takes multiple cycles to execute. To improve the throughput processors use a multi-stage pipelined execution unit. Each stage in the pipeline takes one cycle. The stages in a general pipelined execution unit are shown in Figure 2.5.

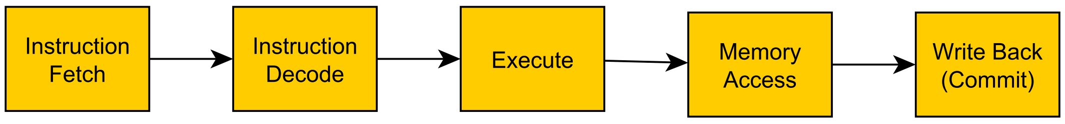


Figure 2.5.: Stages in a Pipelined Execution Unit

The first instruction is *fetched* into the pipeline in the first stage. While this instruction is being *decoded* in the next stage, the consecutively next instruction is *fetched*. The *execution* of the instruction is done in the third stage. The fourth stage is used for instructions that need to access memory, such as the load/store. The result of the execution is written back to the destination register in the final stage. In a best case scenario, the cycles consumed in execution of each basic block can be calculated as follows.

$$Cycles = n\_inst + n\_stages - 1 \tag{2.1}$$

where

$$n\_inst = \text{Number of instructions in the basic block}$$
$$n\_stages = \text{Number of stages in the pipeline}$$

However, consecutive instructions may have data and control dependency which will lead to stalling of the pipeline. Data dependency occurs when one instruction use the result generated by the previous instruction as an input. Refer to the example 2.4. The result of the **add** instruction will be written to **r0** in the final stage in the pipeline. Meanwhile, the next instruction **mul** needs the value of **r0**. The execution of the **mul** instruction will be stalled until, the result of the **add** instruction has been written in **r0**.

```
806c:    ...
8070:    add    r0, r1, r2
8074:    mul    r3, r0, r4
8078:    ...
```

Example 2.4: Illustration of Data Dependency among instructions

Some operations like floating point arithmetic, need more than one cycle to execute. The execution unit for such operations is implemented outside the pipeline. Control dependency occurs when an instruction needs a resource, which is being used by the previous instruction. Again, execution of the subsequent instruction is stalled until the resource is freed by the previous instruction.

To estimate the cycles spent in execution of a basic block, the structure of the pipeline and effects due to Data and Control Dependencies are taken into consideration. Instructions in each basic block are parsed sequentially, and dependencies are identified. The actual cycles spent in execution on the pipeline, are estimated by adding appropriate penalties for pipeline stalls to the equation 2.1.

In this step, the latency in fetching data from the memory is ignored. This will be taken into account in section 2.6. Also, it is assumed that the pipeline is empty at the beginning of each basic block. This may not be the case always, and will be considered in section 2.5.1.

The total number of cycles as calculated from the simulation is annotated to the corresponding basic block in the source code, as illustrated in Example 2.5. 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.

```c
unsigned long long execCycles = 0;

void foo()
{
    ...
    for(i=0; i<20; i++)
    {
        execCycles += 24; // Cycles spent in execution of this basic block
        ...
    }
    ...
}
```

Example 2.5: Annotation for Execution Cycles

2.5.1. Branch Prediction

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

```
1  806c:    ...
2  8070:    mov    r0, #0
3  8074:    ...
4  ...
5  808c:    add    r0, #1
6  8090:    cmp    r0, #20
7  8094:    blt    8074
8  8098:    ...
```

Example 2.6: Implementation of a loop using Conditional Branching Instructions

Conditional branching is implemented in binary code as a compare instruction followed by a conditional branch instruction. Example 2.6 shows the implementation of a loop. The branch on line 7 is taken depending on the result of the compare instruction. In a pipelined execution unit, 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 2.6.

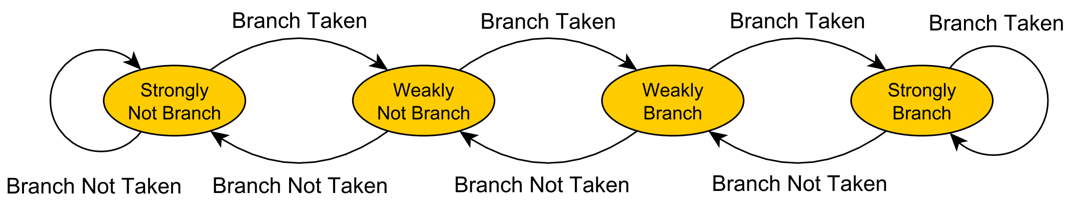


Figure 2.6.: State Machine Diagram implemented in the Branch Prediction Unit

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.

```
/**
 * @brief Function to simulate Branch Prediction Algorithm.
 *
 * @param Start Address of the basic block
 * @param End Address of the basic block
 *
 * @return True, if branch was predicted
 *         False, if branch was not predicted
 */
unsigned int enterBlock(unsigned long startAddress,
                        unsigned long endAddress);
```

Snippet 2.1: API offered by Branch Prediction Simulator

A Branch Prediction Simulator has been developed which uses the same algorithm. It offers the API as shown in Snippet 2.1.

The function is called at the beginning of each basic block. The start and end address of the basic block is passed as a parameter to the simulator. This information is extracted from the binary code. 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 are subtracted from **execCycles**. The instrumentation is done as shown in Example 2.7.

```
unsigned long long execCycles = 0;

void foo()
{
    ...
    for(i=0; i<20; i++)
    {
        execCycles += 24; // Cycles spent in execution of this basic block
        execCycles -= (enterBlock(0x8040, 0x8080) ? 4 : 0);
        ...
    }
    ...
}
```

Example 2.7: Instrumentation for simulating Branch Prediction Unit

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. The source code is instrumented to invoke the Cache Simulator at run-time. In the following sections details about the Cache Simulator have been discussed, followed by the process to instrument the code.

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 API mentioned in snippet 2.3 to simulate different types of memory accesses.

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

```
/**
 * @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);
```

Snippet 2.2: API provided by Cache Simulator

block in bytes as parameters. Number of cycles spent in performing the memory access is returned by the cache simulator. The return value from the cache simulator is accumulated in a global variable **memAccessCycles**.

```
struct csim_result_t
{
    unsigned long long L1Hits;
    unsigned long long L2Hits;
    unsigned long long L2Misses;
    unsigned long long prefetches;
};
```

Snippet 2.3: Data Structure for storing detail statistics from Cache Simulator

In addition, the Cache Simulator provides detailed statistics. These results are stored by the Cache Simulator in an object of type **struct csim_result_t**. The pointer to the object is passed to the Cache Simulator as the third parameter in the API. This data will be useful for estimating power consumption, as will be discussed later.

The source code is instrumented, as illustrated in example 2.8, to create global variables to record the result of cache simulation.

```c
unsigned long long execCycles = 0;
unsigned long long memAccessCycles = 0;
struct csim_result_t csim_result;

void foo()
{
    ...
```

Example 2.8: Global Variables declared for results from Cache Simulation

2.6.1.1. Cache Features and Parameters

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
- Data Prefetching 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 like size and associativity to analyse the impact on performance, without having to instrument the code again.

Details about the hierarchy and features that have been implemented in the Cache Simulator to model the target processor used for testing have been discussed in Section 3.1.4.

2.7. Instrumentation for Instruction Access

For estimating cycles spent in fetching the instructions from memory, instrumentation 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 by static analysis. 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, are the 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. The instrumentation is illustrated in Example 2.9.

```c
unsigned long long execCycles = 0;
unsigned long long memAccessCycles = 0;
struct csim_result_t csim_result;

void foo()
{
    ...
    for(i=0; i<20; i++)
    {
        execCycles += 24; // Cycles spent in execution of this basic block
        execCycles -= (enterBlock(0x8040, 0x8080) ? 4 : 0);
        memAccessCycles += simICache(0x8040, 32, &csim_result);
        ...
    }
    ...
}
```

Example 2.9: Instrumentation for simulating Branch Prediction Unit

2.8. Instrumentation for Data Access

The instrumentation in the example shown in Section 2.1 was simplified for illustration. It had a flaw which will result in inaccuracy of estimation. To simulate load operation for fetching elements of **array**, the address of **array** 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. The host machine in our test setup uses 8 Bytes for **long int** and the target machine uses 4 Bytes for the same data type. When accessing elements of an array of **long int**s using host addresses, twice the number of cache-misses will be reported than what is expected on the target device.

The address for each load/store instruction needs to be resolved for the target processor. This can not be done by static analysis. The technique to do this is inspired from research published in [4] and is called Memory Access Reconstruction.

Multiple steps are involved in this process. These steps are described in the following sections.

2.8.1. Resolve address of each variable

The address of each variable used in the program is extracted. This information is useful in later steps. A program may use different types of variables. The technique to resolve the address of each type of variable has been described in this section.

2.8.1.1. 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 2.10.

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

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

Example 2.10: Instrumentation to resolve address of Global Variables on Target Device

2.8.1.2. 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. Instrumentation is done to keep a track of the Stack Pointer, as shown in Example 2.11. A global variable **CSIM_SP** is declared and initialized with the initial value of the stack pointer. The size of stack frame for each function is extracted by static analysis of the binary. The value of **CSIM_SP** is incremented at the beginning of each function by the size of the stack frame of the function, and decremented before the function returns.

The relative address of a local variable is also annotated in the code. The physical address of the variable at run-time is calculated by adding the relative address to the value of **CSIM_SP**.

```c
unsigned long CSIM_SP = 0x1ff28; // 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 2.11: Instrumentation to resolve address of Local Variables on Target Device

2.8.1.3. Dynamically Allocated Memory

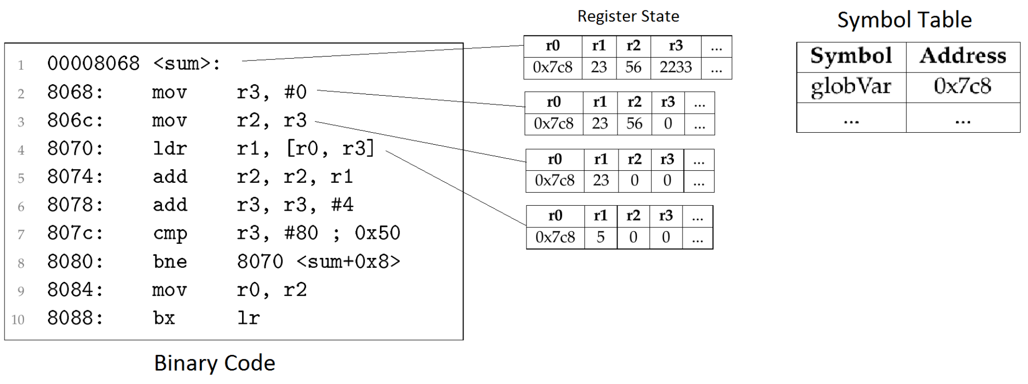
Dynamically Allocated Memory is stored in the Heap Section. To allocate and free heap memory, the application uses API provided by system libraries. The memory allocation algorithm of the target system needs to be emulated at run-time of simulation, to resolve physical addresses of dynamically allocated memory. This approach is discussed in [4]. 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.

2.8.2. 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 functional 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 a function is called, the state of the registers is stored and will be used as the initial state when simulating the function. This is done to keep track of function parameters sent as values in registers. When parameters are sent in the stack, this approach will fail. Each function is simulated once and subsequent calls to the function are ignored.

The illustration in Figure 2.7 shows how this works. The code from the simple example in Section 2.1 has been used for illustration. The binary code of the **sum** function is



```
1  00008068 <sum>:
2  8068:    mov    r3, #0
3  806c:    mov    r2, r3
4  8070:    ldr    r1, [r0, r3]
5  8074:    add    r2, r2, r1
6  8078:    add    r3, r3, #4
7  807c:    cmp    r3, #80 ; 0x50
8  8080:    bne    8070 <sum+0x8>
9  8084:    mov    r0, r2
10 8088:    bx     lr
```

r0	r1	r2	r3	...
0x7c8	23	56	2233	...

r0	r1	r2	r3	...
0x7c8	23	56	0	...

r0	r1	r2	r3	...
0x7c8	23	0	0	...

r0	r1	r2	r3	...
0x7c8	5	0	0	...

Symbol	Address
globVar	0x7c8
...	...

Figure 2.7.: Illustration of the Partial Functional Simulator used to extract address of each load/store instruction in the binary code

shown on the left. On the right, the symbol table shows the address of the global variable, **globVar**.

The state of the registers is shown when the function was called. Address of the **globVar** was sent as a parameter to the function in register **r0**. The values in the rest of the registers are arbitrary. The state of the registers is updated after parsing each instruction in the code. On line 4, a load instruction is seen. The address in **r0** is indexed by the value in **r3**, and the data is read into **r1**. The address belongs to global variable **globVar**.

By doing this 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. Addresses of Global and Static Variables were extracted in the previous stage. The variable being accessed can be 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 for register spilling. The load/store instruction could be associated with this operation. This can be distinctively identified, if no local variable is located on this address.

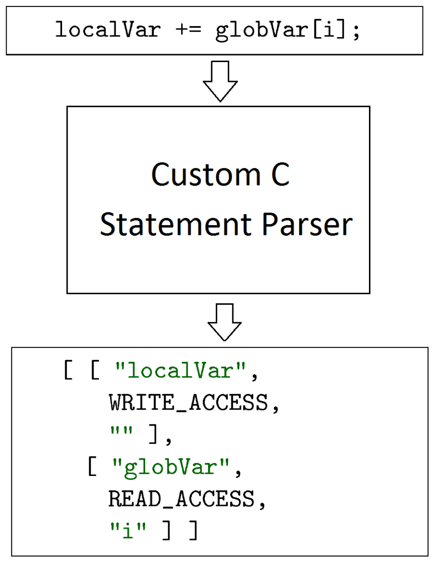
The memory accesses performed in each basic block in the binary code are recorded. Instrumentation for simulating these accesses will be done in the corresponding basic blocks in the source code.

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.

2.8.3. Parse Source Code

In the previous step, load/store instructions for register spilling were distinctly identified. Instrumentation for register spilling is straight-forward. For each load/store instruction corresponding to register spilling in a binary basic block, data cache access is simulated in the mapped basic block in the source code.

However, instrumentation of load/store instruction for variables is a more complicated. Note, that in the previous step only the variable being accessed by each load/store instruction were identified. In the example shown above **globVar** is an array of integers. Elements of the array are accessed in a loop by adding an index to the base address of the array. To correctly instrument the Data Cache Access, the index must be known. This information can only be extracted by parsing the source code.



```
localVar += globVar[i];
```

```
[ [ "localVar",
    WRITE_ACCESS,
    "" ],
  [ "globVar",
    READ_ACCESS,
    "i" ] ]
```

Figure 2.8.: Block Diagram for a custom C parser, that parses a statement and returns information about each variable accessed in the statement.

In this stage, the focus is to identify the statement in source code which causes memory access. By parsing this statement, additional information will be extracted to accurately instrument the code. 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. Refer to figure 2.8.

For each basic block in binary code, the mapped basic block in the source code is parsed. One to one mapping between each load/store operation and a statement in the source code is extracted. Annotation for the data access is performed.

Annotation for simulating access to a global variable is straight-forward. The index

```
1  unsigned long CSIM_SP = 0x1ff28; // Initial Value of SP
2  
3  unsigned long long memAccessCycles = 0;
4  struct csim_result_t csim_result;
5  
6  unsigned int globVar[20];
7  unsigned long long globVar_addr = 0x7c8; // Address of Global Variable
8  
9  void foo()
10 {
11     unsigned int localVar;
12     unsigned long localVar_addr = 0x08; // Address relative to SP
13 
14     CSIM_SP = CSIM_SP + 0x16; // Increment by size of stack frame
15 
16     ...
17     localVar += globVar[i];
18     memAccessCycles += simDCache(globVar_addr + i * sizeof(int),
19                                  READ_ACCESS,
20                                  &csim_result);
21     memAccessCycles += simDCache(CSIM_SP + localVar_addr,
22                                  WRITE_ACCESS,
23                                  &csim_result);
24     ...
25 
26     CSIM_SP = CSIM_SP - 0x16; // Decrement by size of stack frame
27     return;
28 }
```

Example 2.12: Instrumentation for simulating Branch Prediction Unit

expression, if any, extracted from the source code is added to the base address of the global variable in the target processor. For local variables, the value of **CSIM_SP** is added to the relative address to extract the base address of the local variable. The index, if any, is then added to this base address. Example 2.12 shows instrumentation of a local and global variable. The highlighted statement on line 17, causes a read operation on an element of array **globVar** and a write operation to **localVar**. Appropriate instrumentation is shown between lines 18 and 23.

2.8.3.1. Handling of pointers sent as function parameters

In the simple example in Section 2.1, pointer to an array is passed as a parameter to the function. By simulating the binary code in Section 2.8.2, access to global variable **globVar** was identified. However, the function **sum** may be called with a pointer to any other array. Special handling needs to be done for such cases, when pointers are sent as function parameters.

The approach used in this project, is to modify the signature of each function that takes pointers as arguments. For each pointer argument **ptr**, a new argument is added **ptr_addr**. The calling function is accordingly modified, to provide address of the variable on the target processor. When **ptr** is dereferenced, data cache access is simulated using **ptr_addr**.

TODO: Maybe we need another example here.

2.9. Estimation of Power Consumption

Apart from the speed of execution, the power consumption is a key performance indicator for modern processors. The balance between performance and power is critical, and needs to be maintained. An accurate estimate of power consumption can help architects keep this balance in check.

The approach to estimate the power consumption is inspired from [TODO]. The power consumption of processors mainly comes from dynamic power consumption due to switching activity and static power consumption due to the leakage current. This power consumption can be attributed to various components in the processor. These components are in active or idle state, depending on the operations being performed. In idle state, the power consumption is reduced as there is no switching activity. Through careful measurement, accurate estimation of dynamic and static power consumption of each component of the processor can be known.

The Dynamic Power Consumption of a system is a function of the Voltage and Frequency. It is derived by the following formula.

$$P = \alpha C V^2 f \tag{2.2}$$

where

$$\begin{aligned} \alpha C &= \text{Equivalent Switching Capacitance of the circuit} \\ V &= \text{Supply Voltage} \\ f &= \text{Frequency} \end{aligned}$$

The value of equivalent switching capacitance, αC can be measured for each component. Also, the frequency and supply voltage for each component is known. The Static Power Consumption for each component is constant, and can be measured.

Some processors use Dynamic Voltage and Frequency Scaling technique to reduce the power consumption. Certain operating points are defined, and depending on the current load on the components the frequency and voltage is decided. The switching between operating points is handled by firmware, and can be disabled. This feature has not been considered in this project currently, however the approach can be extended to handle this feature effectively.

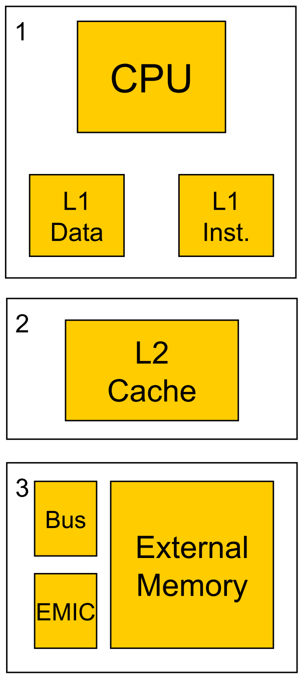


Figure 2.9.: Components of a processor

From information about state of each component at a point in time, the total power being consumed by the processor at that time can be estimated. This information can be extracted from the instrumentation performed in previous steps. An accurate model of components in the processor needs to be created. In this project, the processor has been divided into three components, as shown in Figure 2.9

The CPU and the L1 Cache are collocated on a single die, and generally operate at the same frequency. They are grouped as a single component. This component is active when the instructions are being executed and L1 Miss does not occur. On L1 Miss, the L2 Cache is requested for data. The L2 Cache generally runs on a lower frequency, and is treated as a separate component. The L2 Cache is only active when an L1 Miss occurs. On L2 Miss, the data is fetched from the main memory. The main memory is generally located on an external chip. The memory is linked to the processor, generally though an External Memory Controller on a memory bus. The external memory, along with the interconnect is treated as a third component.

2.9.1. Trace Collection

Rather than average power consumption, it is interesting to see how the consumption changes over the time of execution of the application. In this project, the power consumption is estimated at the granularity of basic blocks. Using statistics generated from instrumentation performed previously, state of each component can be known over the time of execution of the basic block. The power consumed over the period of executing of this basic block can thus be calculated.

The CPU remains in switching state for **execCycles** and in idle state for **memAccessCycles**. Note, that when the data is fetched from L1 Cache, the CPU is not put idle state. Hence, grouping of L1 Cache with the CPU is justified.

The Cache Simulator can tell whether the data was fetched from either of the caches or from the memory. The number of cycles spent in each access from L2 cache is constant. The L2 cache is in active state when an L1 Miss occurs. The data is looked up in the L2 cache. If data is not found, L2 Miss occurs and data is fetched from Memory. The memory is also in active state when data is prefetched by the processor. The Cache Simulator provides detailed statistics of number of L1 Hits, L2 Hits and L2 Misses and prefetches.

For estimating power consumption, this data is collected at the basic block gralularity. The API provided by the Power Estimation Utility is shown in Snippet 2.4

```
/**
 * @brief API to estimate the power
 *
 * @param Name of the current Basic Block
 * @param Number of cycles spent in active state of CPU
 * @param Number of cycles spent in fetching data from memory
 * @param Number of L2 Accesses = L2 Hits + L2 Misses
 * @param Number of memory accesses = L2 Misses + Prefetches
 *
 * @return Amount of Energy spent in the period of time in uJ.
 */
double estimate_power(unsigned long long execCycles,
                      unsigned long long memAccessCycles,
                      unsigned long long L2_Access,
                      unsigned long long memAccesses)
```

Snippet 2.4: API provided by Power Estimation Utility

The API is called at the end of each basic block. The current value of global variables **execCycles** and **memAccessCycles** is passed as parameter. The utility keeps track of previously reported cycles, and calculates the cycles spent in each activity in the current block. Similarly, the current value of sum of number of L2 Hits and number of L2 Misses is passed as the fourth parameter. Current value of sum of number of L2 Misses and number of prefetches is sent as the fifth parameter.

Using this data, the Power Estimation Utility calculates the total power consumed over this period. The data is written to an output file which can be plotted and analysed.

Example 2.13 illustrates the instrumentation for estimating power consumption.

```
unsigned long long execCycles = 0;
unsigned long long memAccessCycles = 0;
struct csim_result_t csim_result;

void foo()
{
    ...
    for(i=0; i<20; i++)
    {
        ...
        estimate_power(execCycles,
                       memAccessCycles,
                       csim_result.L2_Hits + csim_result.L2_Misses,
                       csim_result.L2_Misses + csim_result.prefetches);

    }
    ...
}
```

Example 2.13: Instrumentation for estimating Power Consumption

3. Test Hardware

In this chapter, the features of the target processor used in testing have been described, along with details about how these features were taken into account in simulation.

3.1. Target Processor

The target processor uses an ARM Cortex A5 core. The salient features are as follows,

- Single Core Processor
- In-order Execution Pipeline
- Dynamic Branch Prediction
- Separate L1 Cache for Data and Instructions
- External L2 Cache. Single Cache for Data and Instructions.

3.1.1. Instruction Pipeline

The ARM Cortex A5 Core has an 8 stage pipeline. The stages in the pipeline are shown in Figure 3.1.

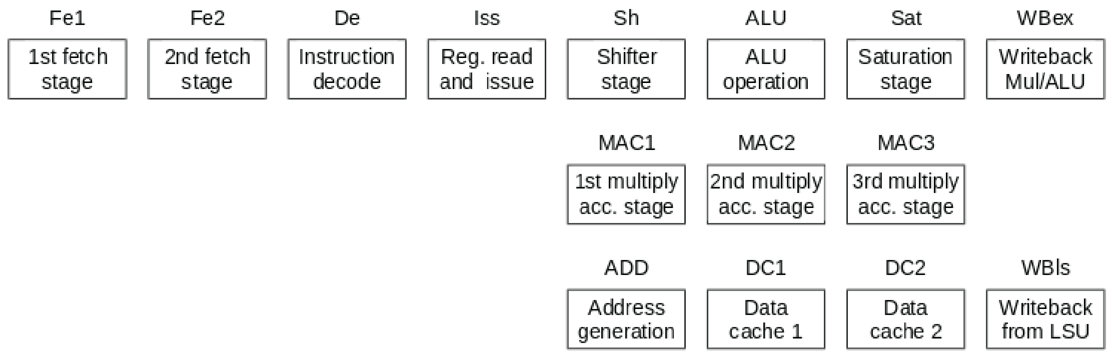


Figure 3.1.: Pipeline Stages in ARM Cortex A5 Core [1]

The first 4 stages are common for all instructions. The next stages are specialized for different types of instructions to reduce interlocking. The Shifter, ALU and Saturation stages are used for Arithmetic and Logical Instructions. The multiplication instruction are executed in 3 multiply accumulate stages. For load store instructions, a separate pipeline is implemented. The pipeline stalls when L1 cache miss occurs and data must be fetched from lower levels of cache or main memory. The result is written back to registers in the last stage.

3.1.2. Interlocking

A strong understanding of interlocking behaviour in the Cortex A5 pipeline is important to estimate the number of cycles spent in execution of each basic block.

Instructions can have complex dependencies. An accurate estimate of number of cycles spent in execution of instructions is difficult to achieve using static analysis. The ARM Reference Manual [1] states that for precise timings a cycle-accurate model of the processor must be used. The approach to calculate the cycles spent in pipeline stalls is described here in brief. This approach, if incorrect could introduce inaccuracy in estimation.

For each instruction, the result of the execution will be written to the destination register in the last stage. The next instruction which needs the result will have to wait. To reduce the number of pipeline stalls, results are forwarded to other stages of the pipeline before writing to the register.

Result Latency Result Latency is the number of cycles before the result of the instruction is available to be used by the next instruction.

```
LDR R1, [R2]                ;Result latency three
ADD R3, R3, R1              ;Register R1 required by ALU
```

For above sequence of instructions, the pipeline will be stalled for 3 cycles.

Early Register An Early Register is a register that is needed at the start of Sh, MAC1 or ADD stage (refer to Figure 3.1) in execution. One more cycle is added to the result latency of the previous instruction producing this register for interlock calculations.

```
LDR R1, [R2]                ;Result latency three
ADD R3, R3, R1  LSL#6       ;plus one, Register R1 is required by Sh
```

In the above example, the value of R1 is needed by the ADD instruction in the Shifter Stage. The pipeline will be stalled for 4 cycles.

Late Register A Late Register is not required until the start of the ALU, MAC1 or DC1 stage for the second execution. One cycle is subtracted from the result latency of the previous instruction producing this result for interlock calculations.

```
LDR R1, [R2]                ;Result latency three
ADD R3, R3, R1, R4 LSL#5    ;minus one, Register R1 is a Late Reg
```

In the above example, the pipeline is stalled for only 2 cycles. While the LDR instruction is being executed, the value in R4 is being shifted. After waiting for 2 more cycles, the value of R1 is made available, and execution can proceed.

The binary instructions in each basic block are parsed sequentially. For each instruction, the early and late registers, and corresponding dependencies are identified. Penalties are appropriately added for the pipeline stalls. The results corroborate that the approach is fairly accurate. The accuracy can be further improved by creating a cycle accurate model of the processor pipeline, and simulating the instructions on this model. This will significantly increase the complexity of the approach and the time needed in instrumentation.

3.1.3. Branch Prediction Unit

The Branch Prediction Unit uses algorithm described in Section [TODO] to predict the outcome of a conditional branch instruction. Cortex A5 has a 125 entry Branch History Table. To simulate the effect of Branch Prediction on performance, a branch prediction simulator has been developed.

3.1.4. Cache Hierarchy

ARM Cortex A5 uses separate L1 Cache for Data and Instructions. An interface is provided to use an L2 Cache. On the test setup, an external unified L2 Cache has been used. The parameters of each cache are shown in Table 3.1. In this section, the details and features of the caches are discussed.

	Size	N-way	Cache Line Size
L1 D Cache	32 KB	4	32 B
L1 I Cache	32 KB	2	32 B
L2 Cache	512 KB	16	32 B

Table 3.1.: Size of Caches

The L1 Caches use a Pseudo Random Replacement Policy, while the L2 Cache used Round Robin Replacement Policy. Cortex A5 uses Exclusive Cache Mode. In this mode, a given address is cached in either L1 Cache or in L2 Cache, but not in both. This has the effect of increasing the usable space and efficiency. The data is loaded from memory into L1 Cache. On eviction from L1, the data is stored in the L2 Cache.

Cortex A5 also supports prefetching of data. The addresses fetched from memory are analyzed and spatial locality of accesses is identified. Data which is highly probable of being used by the application is prefetched in the L1 Cache, thereby saving cycles spent in fetching data from memory.

These features have been implemented in the Cache Simulator to accurately model the Cache. The results corroborate the accuracy in estimation of the number of cache misses occurred.

4. Results

4.1. Test Setup

The general technique described in Chapter 2 was implemented for estimating performance and power consumption of a target processor based on the ARM Cortex A5 core. Details about the target hardware have been briefly discussed in Chapter 3.

The Cortex A5 supports a Performance Monitoring Unit. A cycle counter can be used to calculate the cycles spent in executing the code. Two other counters can be programmed to count various other events. This functionality was used to extract the results from the actual hardware. The accuracy of estimation was calculated by comparing the results from the simulation.

Apart from the cycle count, following results were extracted using the programmable counters.

- Number of Cache Misses
- Number of Data Prefetches
- Number of Branch Mispredictions

A number of benchmark applications were simulated. A brief description of these applications, and detailed analysis of the results for each have been discussed in the following Sections.

4.2. Sieve Benchmark Application

This application

5. Conclusion

Appendices

A. Source Code to ISC Conversion

The approach to convert Source Code to Intermediate Source Code has been inspired from [3]. The Retargettable Back-Annotator (RBA) is made available by [3] under the new BSD license. This tool performs instrumentation, and is customized for the PowerPC architecture. The python code to translate Intermediate Code in GIMPLE to ISC has been reused from this tool.

The motivation behind this approach was covered in detail in Section [TODO]. In this section, some brief examples of the ISC code are shown to familiarize the reader with ISC, and illustrate the benefit of converting Source Code to ISC.

Example A.14 shows a snippet of code from sha benchmark application from the WCET suite [2]. This is a clone of the standard **memcpy** function, and has been chosen for this illustration because of its simplicity.

```c
void* my_memcpy(void* dest, const void* src, size_t count) {
  char* dst8 = (char*)dest;
  char* src8 = (char*)src;

  while (count--) {
      *dst8++ = *src8++;
  }

  return dest;
}
```

Example A.14: Source Code of a function to copy memory content

Example A.15 shows the Intermediate Source Code generated by the tool. It can easily be verified, that the ISC has same functionality as the source code.

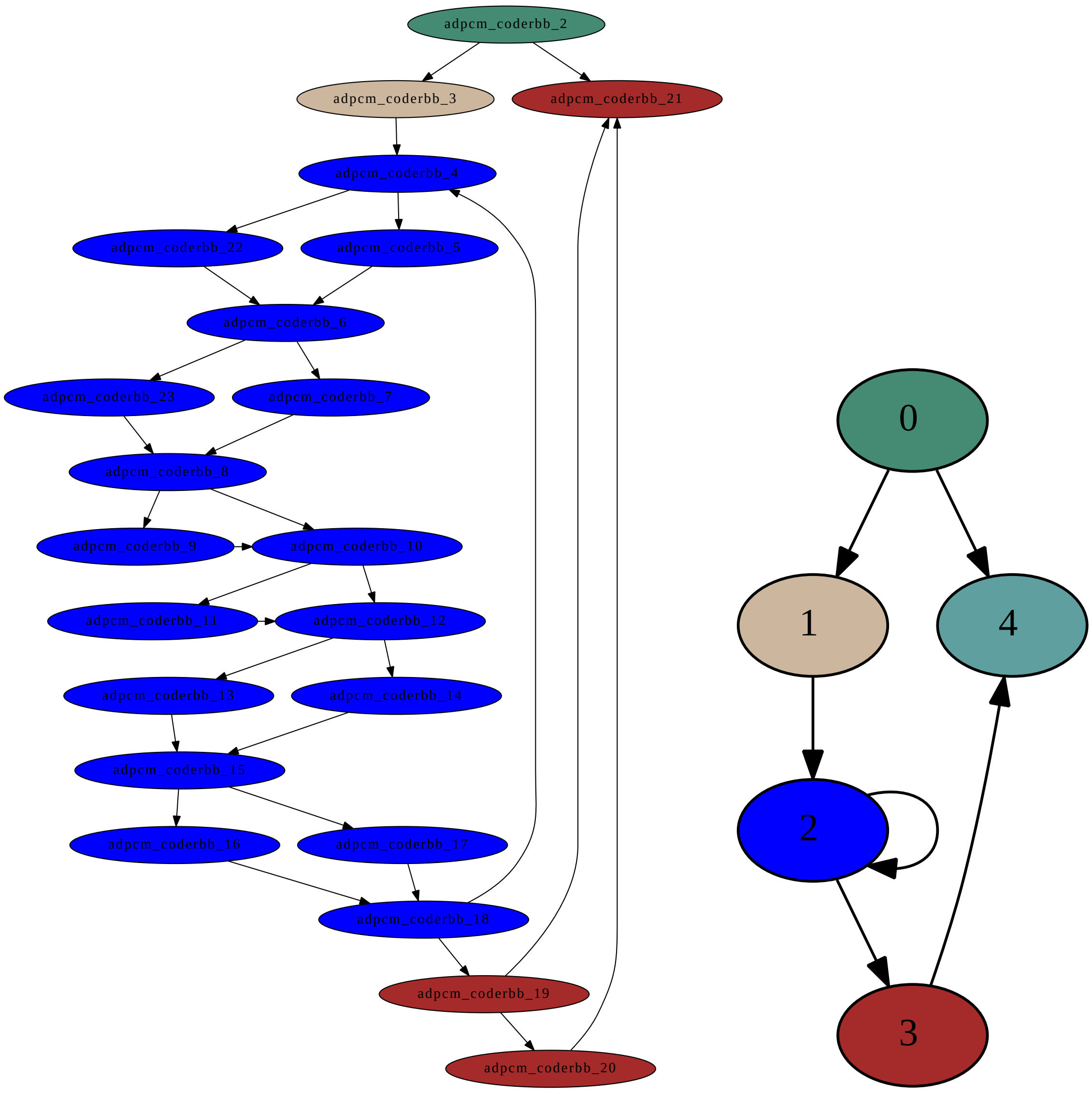
Note, that in the ISC the basic blocks are already marked by labels. The **while** loop in the source code has been implemented as an **if-else** construct and **goto** statements in the block labelled **my_memcpybb_3**. The Control Flow graph is easy to extract from the Instrumented Source Code.

```
void* my_memcpy(void* dest, const void* src, size_t count) {
  uintptr_t ivtmp_23;

my_memcpybb_2:
//  # PRED: ENTRY [100.0%] (fallthru,exec)
  if (count != 0)
    goto my_memcpybb_5;
  else
    goto my_memcpybb_4;
//  # SUCC: 5 [91.0%] (true,exec) 4 [9.0%] (false,exec)

my_memcpybb_5:
//  # PRED: 2 [91.0%] (true,exec)
  ivtmp_23 = 0;
//  # SUCC: 3 [100.0%] (fallthru)

my_memcpybb_3:
//  # PRED: 3 [91.0%] (true,exec) 5 [100.0%] (fallthru)
  *(char *)((uintptr_t)dest + (uintptr_t)ivtmp_23) = *(char
      *)((uintptr_t)src + (uintptr_t)ivtmp_23);
  ivtmp_23 = ivtmp_23 + 1;
  if (ivtmp_23 != count)
    goto my_memcpybb_3;
  else
    goto my_memcpybb_4;
//  # SUCC: 3 [91.0%] (true,exec) 4 [9.0%] (false,exec)

my_memcpybb_4:
//  # PRED: 3 [9.0%] (false,exec) 2 [9.0%] (false,exec)
  return (uintptr_t)dest;
//  # SUCC: EXIT [100.0%]

}
```

Example A.15: Intermediate Source Code generated from Example A.14

B. Mapping Algorithm

Figure B.1 shows an example of the mapping generated by the Mapping Algorithm.



(a) CFG from Intermediate Source Code

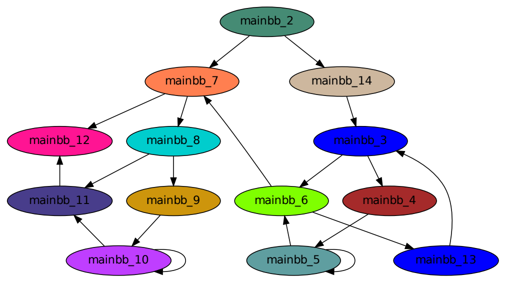
(b) Control Flow Graph (CFG) from optimized Binary Code

Figure B.1.: Illustration of mapping generated by the Mapping Algorithm

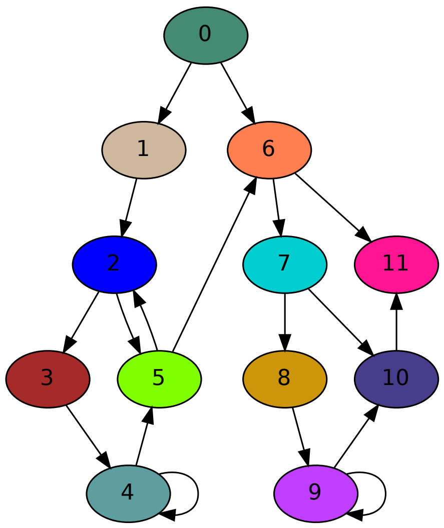
The adpcm benchmark application is used from the WCET suite [2]. Figure B.1a shows the CFG extracted from the Instrumented Source Code of the function **adpcm_coder**. Figure B.1b shows the CFG extracted from the optimized cross-compiled binary code of the same function. The color coding depicts the mapping between basic blocks of the CFGs.

As can be seen, the blue nodes in the ISC were collapsed to a single node in the Binary Code. This was due to Conditional Execution Optimization used by the compiler. Conditional Execution was explained in Section [TODO].

Figure B.2 shows another example of a mapping generated by the algorithm.



(a) CFG from Intermediate Source Code



(b) CFG from optimized Binary Code

Figure B.2.: Illustration of mapping generated by the Mapping Algorithm

For this example, the **main** function in ctop benchmark from WCET suite [2] was used for illustration. In this example, the 2 blue nodes in ISC in Figure B.2a were collapsed to a single node in the Binary. Apart from this, no other blocks were merged. The mapping algorithm was able to accurately map the blocks to each other.

Similarly, for the set of benchmarks used for testing the mapping algorithm was able to extract mapping accurately.

List of Figures

List of Tables

Bibliography

[1] *Cortex-A5 Technical Reference Manual*.

[2] Worst-case execution time (wcet) analysis project.

[3] 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.

[4] Kun Lu, D. Muller-Gritschneder, and U. Schlichtmann. Memory access reconstruction based on memory allocation mechanism for source-level simulation of embedded software. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 729–734, Jan 2013.