FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

Host Compiled Simulation for Timing and Power Estimation

Gaurav Kukreja

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

Host Compiled Simulation for Timing and Power Estimation

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

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

Munich, 15th October, 2014 Gaurav Kukreja

Abstract

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

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

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

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

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

Contents

1 Introduction

1.1 Simulation

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

Simulation is widely used in Hardware Software Co-development. The behaviour of a processor is simulated. It allows Hardware Developers to analyse and validate the performance impacts of design decisions at early stage of hardware development. This allows them to save the crucial effort and cost involved in fabricating hardware.

The widely used approach in this area, is called Instruction Set Simulator (ISS). In ISS the processor micro-architecture is simulated in great detail. Each stage of processor pipeline is simulated along with other building blocks of the processor like Caching and Branch Prediction Units. This approach provides cycle accurate estimates of performance. However, ISS is difficult to develop and slow to execute because of the amount of details that are simulated. ISS is not suitable for simulation of long running software scenarios.

The focus of this research is to enable fast simulation of single core, embedded processors to provide accurate estimation of time and power spent in executing benchmark applications.

1.2 Related Work

Considerable work has been done in this area of research. The techniques developed can be roughly divided into two approaches.

1.2.1 Sampling Based Approach

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

In Sampling Based Approach, certain samples of the execution trace are simulated in detail using Instruction Set Simulation. The rest of the execution is carried out using Functional Simulation at a higher level of abstraction. This leads to a speed up in simulation.

Research in this area mainly focuses on developing procedures to select the samples.

1.2.2 Host Compiled Simulation

Host Compiled Simulation is based on 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.

To accurately estimate the performance of a processor, we must take into consideration the effects due to following optimization techniques used in modern processors.

- Processor Pipelining
- Branch Prediction
- Memory Caching and Prefetching

The instrumentation enables simulation of these

1.3 Focus

1.3.1 Simple Example

1.4 Thesis Outline

2 Background

3 Host Compiled Simulation

Host Compiled Simulation is the current focus areas of research in the space of simulation. It is popular because it is simple to understand and develop compared to other approaches, and can provide highly accurate estimates of performance.

Host Compiled Simulation (HCS) is based on the approach of Source Code Instrumentation. Instrumentation is the technique to modify the source code of an application, so as to extract useful information during the run-time of the application. In Host Compiled Simulation, the source code is instrumented to gain information of the time spent in executing the code on a particular target processor.

The source code of the application is analysed along with the cross-compiled binary code generated by the compiler for the target processor. Time spent in executing each basic block of the binary code is estimated and annotated to the corresponding basic block in the source code. For each memory access in the binary code, time spent in performing the operation on the target processor is estimated and annotated. This annotated time for each basic block and memory access is accumulated in a global variable, and reported at the end of the execution.

The instrumented code is compiled for and run on the Host Machine, hence the name Host Compiled Simulation.

3.1 Simple Example

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

Consider the source code in Listing 3.1. The function *sum* calculates the sum of elements in an array and returns the result. The object dump from the binary code generated by the ARM cross-compiler is shown in Listing 3.2.

To accumulate the number of cycles spent in execution two global variables will be declared, *execCycles* and *memAccessCycles*.

From the binary code, 3 basic blocks can be identified. The first basic block starts from line number 2 to 3, the second is a loop from line 4 to 8 and the third between lines 9 and 10. These blocks can easily be matched to corresponding blocks in the source code.

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

Table 3.1: Mapping of Basic Blocks

For each basic block, we will annotate the time spent in executing the instructions. For simplicity, let us assume that the processor has a single stage pipeline, and each instruction takes 1 cycle to execute. The estimate number of cycles is added to the global variable *execCycles*.

Further, we can see one memory access on line 4, which corresponds to the loading of elements of the array. To estimate the time spent in loading this data, the cache hierarchy of the target system will be simulated. The cache simulation offers an API *simDCache* to simulate data cache access and takes as parameter the address of the data and a flag to signify whether it is a read or write operation. The cache simulator returns the number of cycles spent in performing the memory access. This value is accumulated in the global variable *memAccessCycles*.

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

The instrumented code is shown in Listing 3.3.

```
1   int sum(int array[20])
2   {
3      int i;
4      int sum = 0;
5
6      for (i=0; i<20; i++)
7         sum += array[i];
8
9      return sum;
10  }
```

Listing 3.1: Simple C Code

```
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
```

Listing 3.2: Objdump Code

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

```c

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, READ);
      execCycles += 5;
      memAccessCycles += simICache(0x8070, 40);
   }

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

Listing 3.3: Instrumented Code

3.2 The Approach

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

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

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

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

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

3.3 Mapping between Source and Binary Code

For correctly instrumenting the source code, we need to know the accurate mapping between source code and binary code. This mapping is usually destroyed by the optimization phases of the compilers. The Compiler performs optimizations like partial or full loop unrolling, etc. This makes reconstruction of the mapping a challenging problem.

GDB uses debug information to map each instruction in the binary code to a line in the source code. However this mapping is highly inaccurate, and can not be used. Prominent techniques presented by research papers in the area use Control and Data Flow Analysis to reconstruct the mapping at a Basic Block granularity.

In this project, the mapping algorithm described in [TODO] has been used. The high level source code is first cross-compiled using the compiler for the target processor. The intermediate code that is generated from the compiler front-end is in GIMPLE format. GIMPLE is a simple 3 address representation of the code, which has been optimized with the target independent optimization strategies. Since most of the optimization phases have already been applied to the code, it is expected that the Control Flow of the Intermediate Code is close to that of the binary code. The intermediate GIMPLE code is then converted back into C Code. To convert GIMPLE to C Code, the code from the tool desrcibed in [TODO] has been reused. The generated C Code is referred to as Intermediate Source Code (ISC).

Here after, the Control Flow Graph (CFG) is generated from ISC and the cross-compiled binary. These CFGs are matched. From observation, it has been noticed that the CFGs still differ quite substantially. Special algorithm is required to match these CFGs.

The technique used in the project to match the CFGs is a Graph Matching Algorithm using Depth First Traversal of the graphs along with special handling for particular optimizations.

One such optimization that needs to be handled is merging of branches using Conditional Execution or Predicated Executed. For if-else branches with small code, the compiler optimizes the code by using Conditional Execution Instruction. The compare instruction is followed instructions inside the *then* and *else* branch which are predicated to indicate that the instructions will only be executed depending on the result of the compare instruction. The statements are viewed as one Basic Block in the Binary code, as the branch instructions are eliminated. Each instruction is fed to the processor pipeline and executed, but the results are only written back if the condition evaluates to *true*. This saves the time spent in branching, and reduces the number of pipeline flushes.

To handle this case, the matching algorithm must identifies Basic Blocks in the binary that contain Conditional Execution Instructions, and appropriately maps a branch in the CFG of the source code. A short example below will explain how this works.

Similarly, other optimization techniques need to be handled specifically. Compilers provide an option to the developer to configure the Optimization Level to be used. With each level a set of optimization strategies are applied. Currently, in this project we focus only on O1 level optimization. The code can be later extended to accommodate for other optimizations.

In some cases, using a Graph Matching algorithm is not enough for accurate mapping. For instance, when two branches from a node look exactly the same, like an if-then-else branch shown in the following example, the matching algorithm is unable to identify the *then* branch from the *else* branch. To map this, we use information from GDB.

There might be some corner cases that might occur using this approach. Most corner cases are identified by the tool and an error message is reported to tell the user, that accurate mapping could not be found. Also a graphical representation of the CFGs is presented depicting the mapping generated by the tool, to help the user identify any mistakes that might have occurred. As of now there is no mechanism to utilize user assistance in fixing mapping issues. This feature can be added as an extension.

The Graph Matching Algorithm returns a data structure with CFGs generated for each function from the source code and the binary code, along with the mapping information. Each basic block in source code maps to one block in the binary, but the basic block in binary may be matched to multiple blocks in the source code. This information is passed on to the next steps.

3.4 Instruction Execution Time Annotation

Time information for execution of instructions in the processor pipeline is annoated at Basic Block granularity.

4 Implementation

5 Results

6 Conclusion

List of Figures

List of Tables