Host Compiled Simulation for Timing and Power Estimation

Gaurav Kukreja

Technical University of Munich

gaurav@gauravk.in

October 26, 2014

Overview

Simulation

Simulation is the technique to imitate the behaviour of a system.

- Widely used in Hardware Software Co-development.
- Use cases are performance analysis, functional verification etc.

Simulation: Popular Techniques

Instruction Set Simulation

- Detailed simulation of processor micro-architecture.
- Cycle Accurate estimation of performance.
- Difficult to develop, Very Slow execution.

Functional Simulation

- Simulation at higher level of abstraction. Details of micro-architecture ignored.
- Very fast simulation.
- Focus is Functional Verification. Cannot be used for performance estimation.

Our Focus

A technique for fast simulation of embedded processors that is,

- Easy to Develop.
- Fast to Execute.
- Highly Accurate in Performance Estimation.

Host Compiled Simulation

Host Compiled Simulation

- Based on technique of Source Code Instrumentation.
- Instrumented code run on Host Machine, hence the name.
- Easy to understand, develop and maintain.
- Fast Execution, and accurate results.

Simple Example

```
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 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 2 : Objdump Code

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

Instrumented Code

```
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  }
```

```
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  }
```

```
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 : Instrumented Code

Objective

- Develop a tool for Automatic Instrumentation.
- ARM Cortex A5 based processor as reference target device.
- Bare-Metal Applications.
- Generate Time and Power Consumption Estimates.

Outline of our Approach

- Generate Mapping between Source Code and Binary Code.
- Extract Information from GDB
- Data Cache Simulation
- Instruction Cache Simulation
- Annotation for cycles spent in Pipeline

Mapping between Source and Binary

- Accurate mapping is needed for instrumentation.
- Compiler destroys mapping during optimization phases.
- GDB provides mapping, but highly inaccurate.
- Algorithms use static and dynamic analysis of Control and Data Flow.

Mapping between Source and Binary

In this project, mapping is generated using following steps.

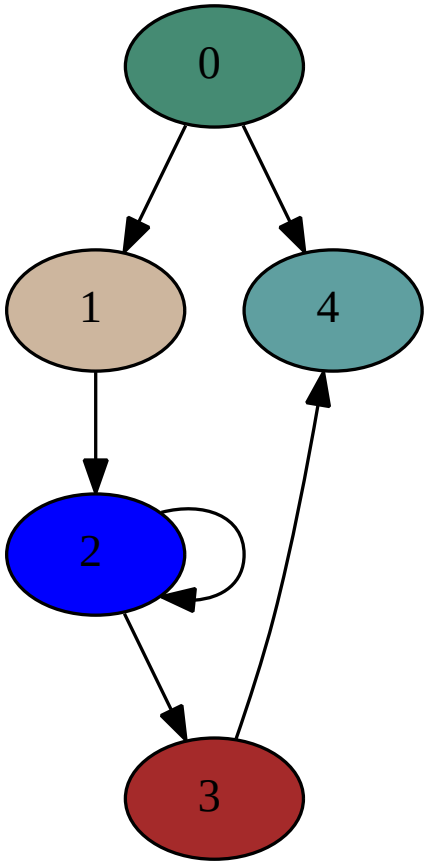
- Cross-Compile Source Code.
- Convert IR Code to C Code (Intermediate Source Code).
- Extract CFG from ISC and Binary Code.
- Map CFGs using Matching Algorithm

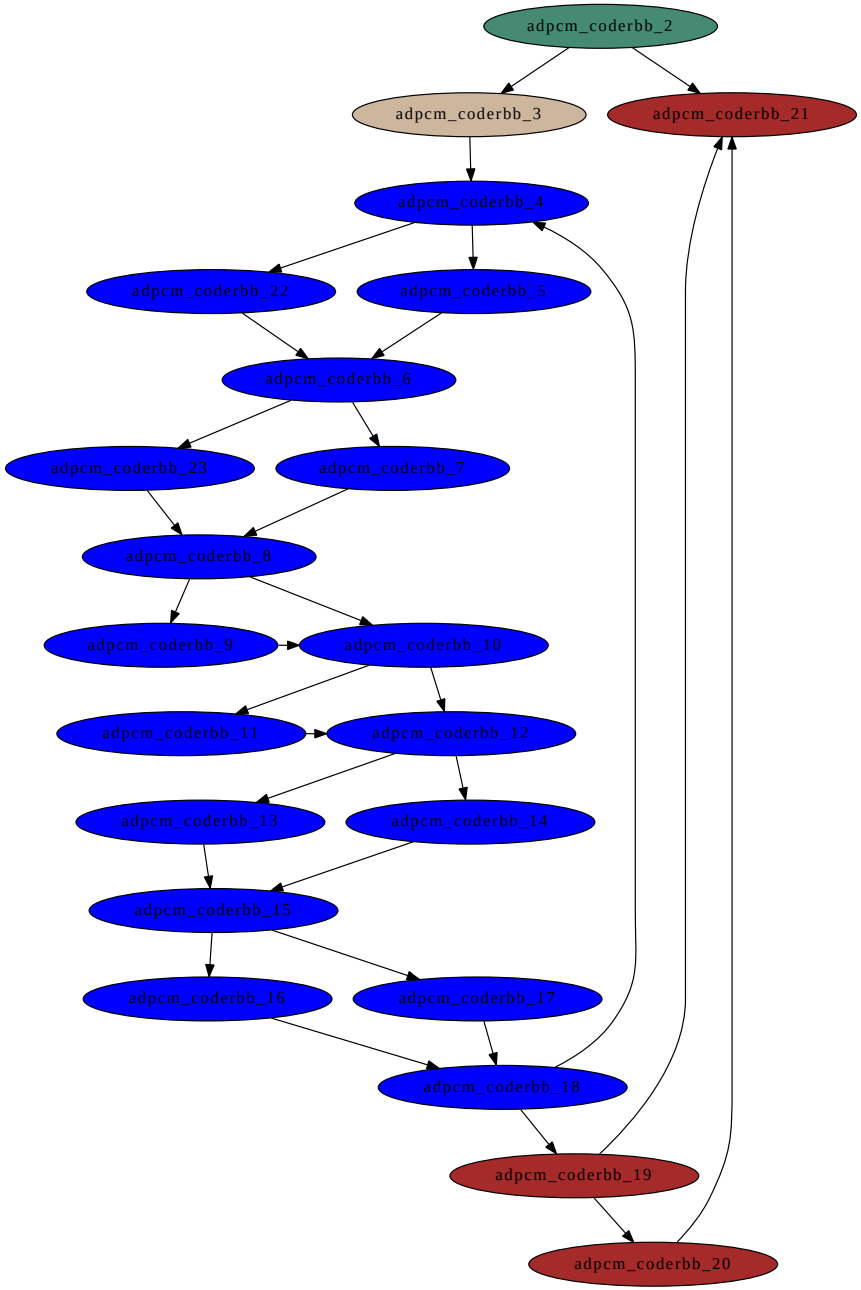
Conversion of IR Code to C Code

- IR Code already contains front-end (processor independent) optimizations. Control Flow closer to Binary Code.
- IR Code is in GIMPLE format.
- GIMPLE Code is converted to C Code.
- The generated code is called Intermediate Source Code (ISC).

Mapping Algorithm

- ISC and object dump of binary code is parsed, and CFGs are extracted.
- Graphs are traversed recursively in Depth First Fashion, to match each branch.
- Special Handling for optimizations that modify Control Flow.
- GDB Debug information in Corner Cases.


0
1
4
2
3


adpcm_coderbb_2
adpcm_coderbb_3
adpcm_coderbb_21
adpcm_coderbb_4
adpcm_coderbb_22
adpcm_coderbb_5
adpcm_coderbb_6
adpcm_coderbb_23
adpcm_coderbb_7
adpcm_coderbb_8
adpcm_coderbb_9
adpcm_coderbb_10
adpcm_coderbb_11
adpcm_coderbb_12
adpcm_coderbb_13
adpcm_coderbb_14
adpcm_coderbb_15
adpcm_coderbb_16
adpcm_coderbb_17
adpcm_coderbb_18
adpcm_coderbb_19
adpcm_coderbb_20

Data Cache Simulation

- For accurately annotating time spent in memory access, data cache must be simulated.
- Cache Simulator to imitate the cache on Target Processor is needed.
- Host Addresses can not be used for Cache Simulation. (???)
- Memory access, as it would occur on the target processor, needs to be simulated.

Cache Simulator

The device used for testing uses ARM Cortex A5. The cache hierarchy used in target processor has been implemented in the Cache Simulator

	Size	N-way	Cache Line Size
L1 D Cache	32 KB	4	32 B
L1 I Cache	32 KB	2	32 B
L2 Cache	256 KB	16	32 B

- Pseudo Random Replacement Policy
- Data Prefetching

Cache Simulator

The Cache simulator offers following API for Data Cache Simulation

```
/**
 * @brief Function to simulate Data Cache Access.
 *
 * @param Address of the memory being accessed.
 * @param True, if access is Read Access.
 *        False, if write access.
 *
 * @return Number of cycles spent in performing access.
 */
unsigned long long simDCache(unsigned long address,
                             unsigned int isReadAccess);
```

Memory Access Reconstruction

For simulation of cache, addresses from Host Machine can not be used. It may lead to inaccuracies because of,

- Memory Alignment differences.
- Different Sizes of Basic Data Types

Let us look at how to reconstruct each memory access, as it would occur on the target processor.

Memory Access Reconstruction

- GDB is used to extract information about each variable used.
- Binary Code is parsed to identify load/store instructions.
- Variable being accessed by the instruction is identified.
- Each Load/Store instruction is matched to an instruction in Source Code that causes the memory access.
- Memory access of the variable is appropriately instrumented.

Extracting information using GDB

- The source code is compiled to run on bare-metal.
- Physical address of each Global Variable can be extracted statically from the binary using GDB.
- Address of local variables, relative to the stack pointer can be extracted using GDB.

This information will later be used.

Identify Load/Store Operations in Binary Code

To identify which variable is being accessed by a load/store instruction,

- The binary code is partially simulated.
- Register State is maintained. Each instruction in binary code is parsed, and registers are updated according to the instruction.
- Branch instructions are ignored, so each instruction is only simulated once.
- For each load/store instruction, the address being accessed can be known.
- Using this address and the information extracted from GDB, the variable being accessed can be identified.

Variables to accumulate statistics

To accumulate the number of cycles spent in execution of the program, teo global variables are declared.

- `execCycles` is used for cycles spent in active state of processor. ie. when instructions are being executed in the pipeline.
- `memAccessCycles` is used for cycles spent in fetching data from memory, when the processor is in idle state and pipeline has been stalled.

```
unsigned long long execCycles;
unsigned long long memAccessCycles;
```

Stack Pointer Simulation

- Global Variable `CSIM_SP` is declared to maintain the value of Stack Pointer.
- `CSIM_SP` is incremented at the beginning of each function, by the size of the stack frame of the function.
- Size of stack frame for each function can be known from GDB.

```
unsigned long CSIM_SP = 0x60c0;
...
void foo() {
    CSIM_SP += 0x30;
    ...
}
```

Address of variables on Target

- For each Global Variable being used, a new global variable is declared to store the address of the variable in the target processor.

```
int array[10];
unsigned long array_addr = 0x7c08;
```

- Similarly, for each local variable a new local variable is declared to store the address of the variable relative to the stack pointer.

```
void foo() {
    int average;
    unsigned long average_addr = 0x8;
}
```

Annotation of Memory Access

To find the line in source code, which causes the memory access operation,

- Each line in ISC is parsed using a C parser.
- The line which causes the load/store operation is identified.
- The variable being accessed may be an array, indexed by a value. This is identified.
- Annotation to perform cache simulation is appropriately added.
- Example ...

Example: Annotation of Memory Access

```
...
for (i=0; i<10; i++) {
    sum += array[i];
    memAccessCycles += simDCache{array_addr + i * 4, True);
}
...
average = sum / 10;
memAccessCycles += simDCache(SP + average_addr, False);
...
```

Cache Simulator

The Cache simulator offers following API for Instruction Cache Simulation

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

Instruction Cache Simulation

Annotation for Instruction Cache Simulation is much simpler.

- For each basic block in binary code, size of the basic block in size is calculated. Start address of the basic block is known.
- Annotation is added in the beginning of the mapped basic block in the Source Code.

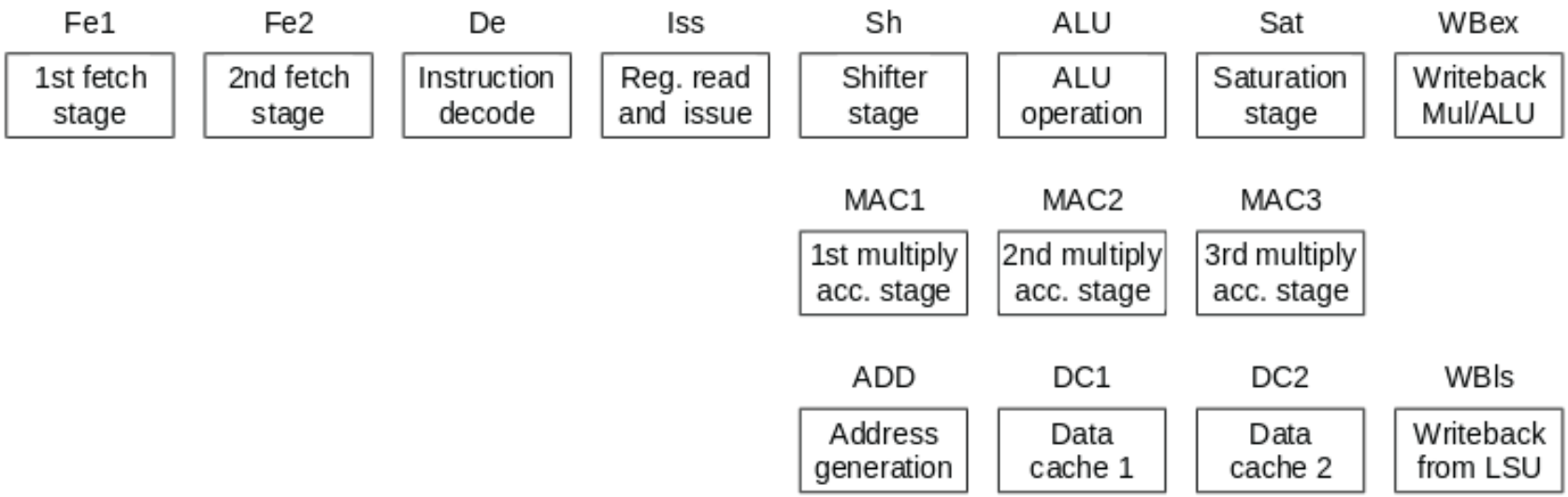
Annotation for Cycles spent in Pipeline

For estimating the cycles spent in pipeline execution, following points need to be considered.

- Pipeline Architecture of the target processor.
- Effects due to Data and Control Hazards, that leads to pipeline stalls.
- Branch Prediction, that prevents pipeline flushes.

Pipeline architecture of ARM Cortex A5

The ARM Cortex A5 has an 8-stage pipeline. The stages are graphically represented below.



Effects due to Data and Control Hazards

To estimate the number of cycles a basic block will take to execute,

- It is assumed, that all data is needed by instructions is available in registers.
- Each instruction in the Basic Block is parsed.
- Initially the pipeline is assumed to be empty.
- Without interlocking each instruction takes 1 cycle to execute.
- Interlocking between instructions is identified, and penalties are added.

Annotation of Cycles spent in Pipeline

- For each basic block in Binary code, annotation is added to the mapped Basic Block in the Source Code.
- The global variable `execCycles` is incremened by the number of cycles.

```
...
for(i<0; i<10; i++) {}
    execCycles += 23;
    sum += array[i];
    ...
}
...
```

Branch Prediction

- Processors use Branch Prediction to reduce the number of pipeline flushes.
- This has a major impact on performance.
- Branch Prediction Unit is emulated to accommodate this effect.

Emulation of Branch Prediction Unit

- Cortex A5 uses a 125 entry Branch History Table, to maintain information whether a previous branch was taken or not-taken.
- For each branch, a 2-bit state information is stored. The states are shown in the State Machine diagram below.
- For each branch instruction seen first time, the BPU sets state to SNB, and assumes that the branch will not be taken.
- The state is updated, depending on whether the prediction was correct or not, as illustrated in the State Machine Diagram.

Annotation for Branch Prediction

A simulator for Branch Prediction has been implemented. It offers following API.

```
/**
 * @brief Function called at the beginning of a Basic Block
 *        to simulate Branch Prediction Unit.
 *
 * @param Start Address of Basic Block being entered.
 * @param End Address of Basic Block being entered.
 *
 * @return True, if branch was correctly predicted.
 *          False, if branch was not correctly predicted.
 */
unsigned int branchPred_enter{unsigned long startAdd,
                              unsigned long endAdd};
```

Annotation for Branch Prediction

- For each basic block in binary code, annotation is added to the mapped basic block in source code.
- Start and End address of basic block in binary code, is passed as parameter.
- Branch History Table is maintained. The function returns True if the branch was correctly predicted.
- If return value is true, penalty previously added is subtracted from `execCycles`.

Example: Annotation for Branch Prediction

```
...
for(i=0; i<10; i++) {
    execCyles += 23;
    execCycles -= (branchPred_enter(0x348, 0x380) ? 7 : 0);
    sum += array[i];
    ...
}
...
```

TODO

Limitations

In some corner cases, the tool may fail to instrument the code.

- Mapping of Source and Binary Code could not be done.
- Load/Store Instruction could not be matched.
- A source line could not be correctly parsed by the C Parser.

In each of such corner cases, the tool proceeds with the rest of the instrumentation. It issues appropriate error and warning messages to assist user in manually fixing these issues.

Usability of tool

- The tool performs as with minimal corner cases, while instrumenting the test benchmark applications.
- For some corner cases, trivial user assistance is needed.
- Effort has been taken, to implement the tool in a way such that it can be easily extended. The code is modular, and well documented.

Test Setup

- For testing the tool, the results have been compared with results from the actual hardware.
- Lauterbach in-circuit debugging tool has been used to run tests and extract results from the hardware.
- Lauterbach can provide exact estimates of the cycles spent in execution. Performance Monitoring Unit in ARM has been used to calculate the number of cache misses at each cache.

Test Results

At the moment, test results for only one benchmark application are available. This is because of contention over hardware resources, and limitation of time.

ADPCM Benchmark

	HCS	Actual	Accuracy
Total Cache Miss	91452	91604	99.99%
Total Cycles	46110394	45594862	99.98%