Host Compiled Simulation for Timing and Power Estimation

Gaurav Kukreja
Bo Wang

Intel Mobile Communications

October 29, 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

Cycle Accurate Simulation

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

Functional Simulation

- High Level of Abstraction.
- Simple to develop, and fast execution.
- 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.

Related Work

Sampling Based Approach

- Small Samples Executed using CAS.
- Rest execution fast-forwarded using Functional Simulation.
- Results Interpolated.
- Inaccurate, Difficult to develop (CAS)

Host Compiled Simulation

Host Compiled Simulation

- Based on technique of Source Code Instrumentation.
- Instrumented code compiled and 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  }
```

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
```

Objdump Code

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  }
```

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
```

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
```

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.

Mapping between Source and Binary

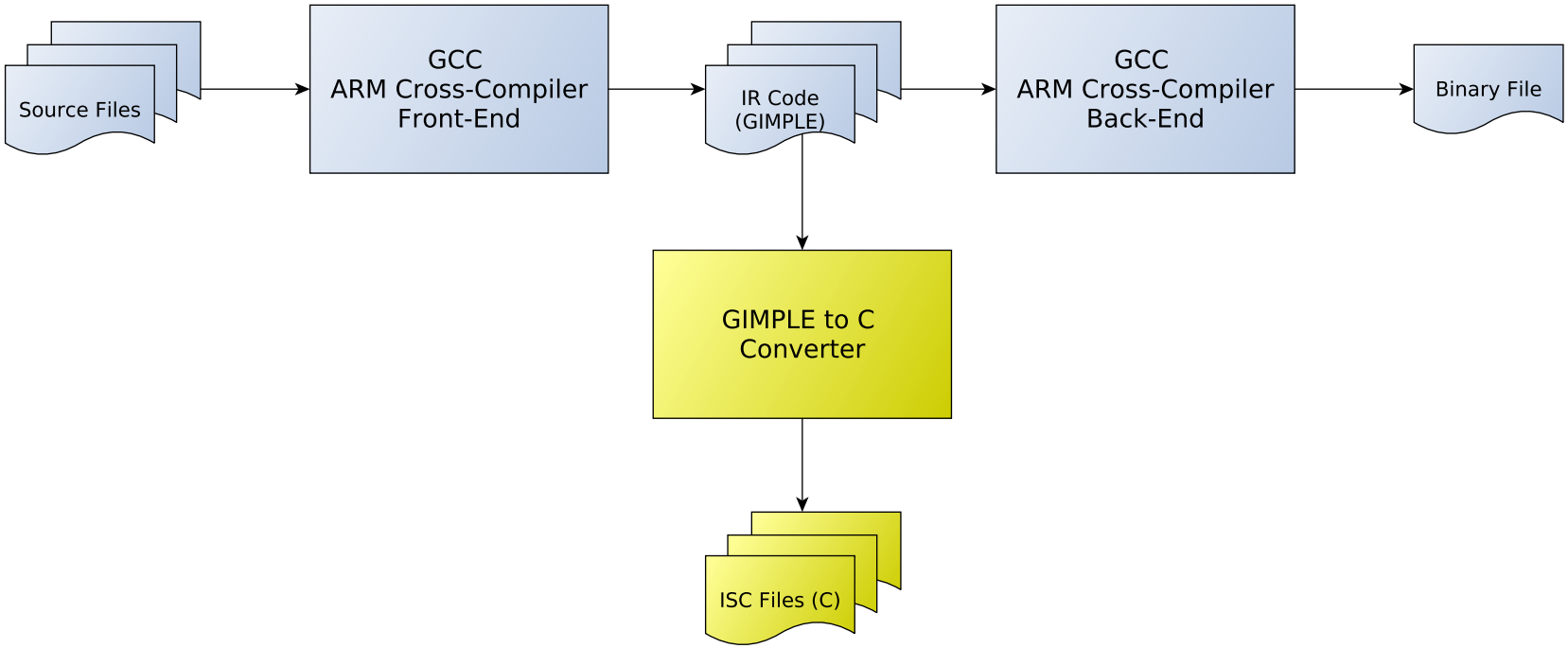
- Very important for accurate instrumentation.
- Compiler destroys mapping during optimization phases.
- GDB provides mapping, but highly inaccurate.
- Control Flow Analysis to generate mapping between Basic Blocks.

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 Mapping Algorithm

Conversion of IR Code to C Code



* Code for GIMPLE to C converter reused from RBA project [TODO]

Mapping Algorithm

- Extract CFG from ISC and Binary Code.
- Graph Matching Algorithm using Depth First Traversal.
- Special handling for each optimization.
- GDB Debug information in Corner Cases.

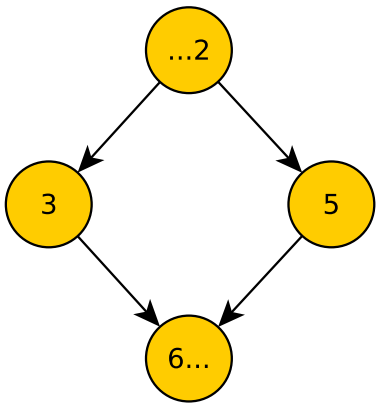
Handling of Compiler Optimization : Conditional Execution

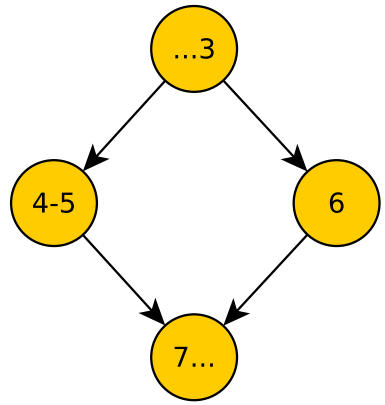
```
1  ...
2  if (a > b)
3      max = a;
4  else
5      max = b;
6  ...
```

Simple C Code

```
1  806c:    ...
2  8070:    cmp    r1, r2
3  8074:    ble    8080
4  8078:    mov    r3, r1
5  807c:    b      8084
6  8080:    mov    r3, r2
7  8084:    ...
```

Unoptimized Assembly Code





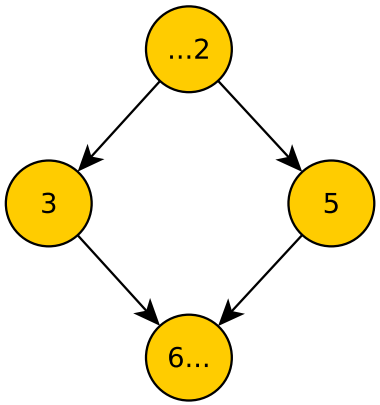
Handling of Compiler Optimization : Conditional Execution

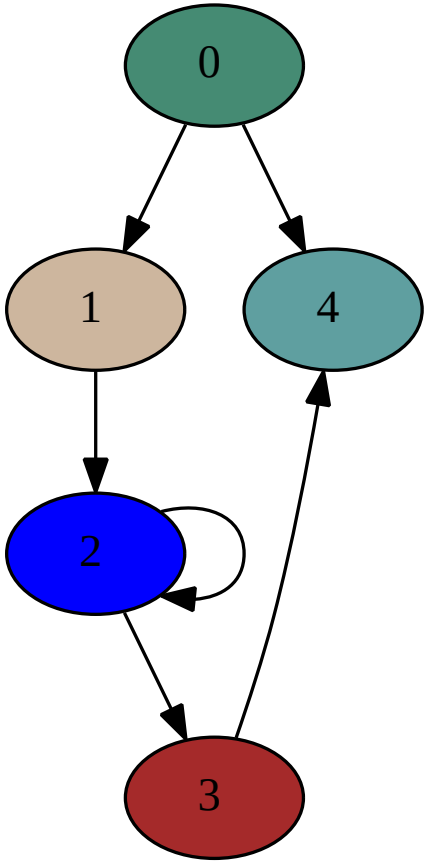
```
1  ...
2  if (a > b)
3      max = a;
4  else
5      max = b;
6  ...
```

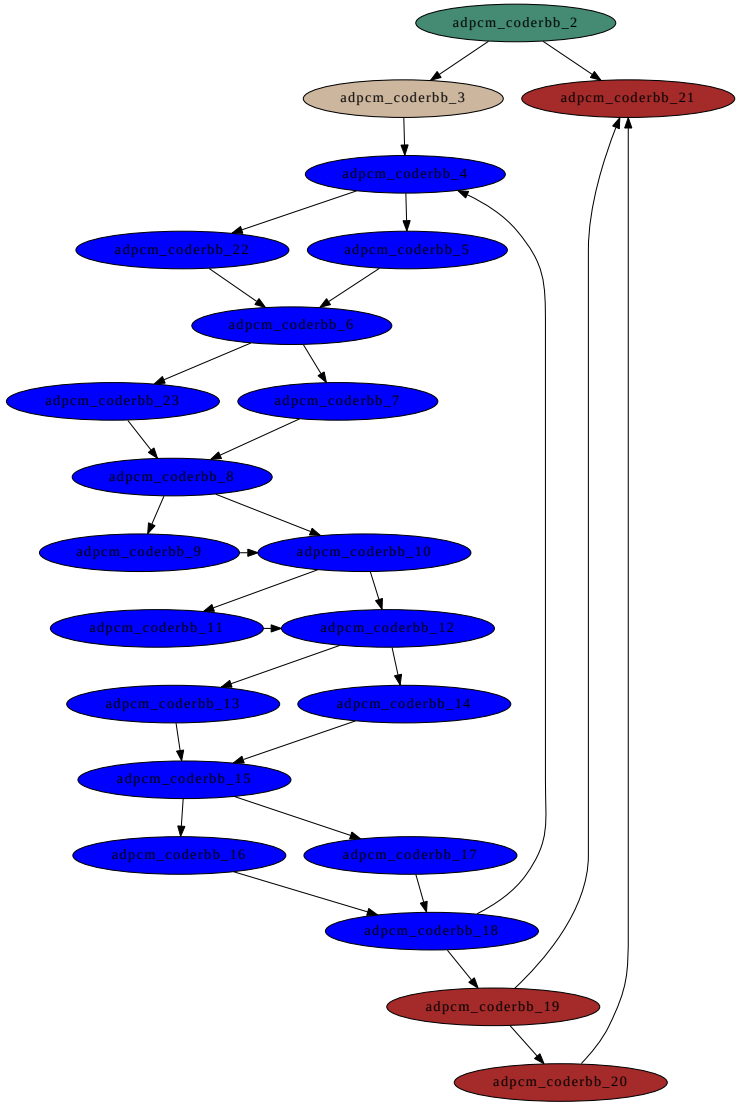
Simple C Code

```
1  806c:    ...
2  8070:    cmp     r1, r2
3  8074:    movgt   r3, r1
4  8078:    movle   r3, r2
5  807c:    ...
```

Optimized Assembly Code




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

Annotation for Cycles spent in Pipeline

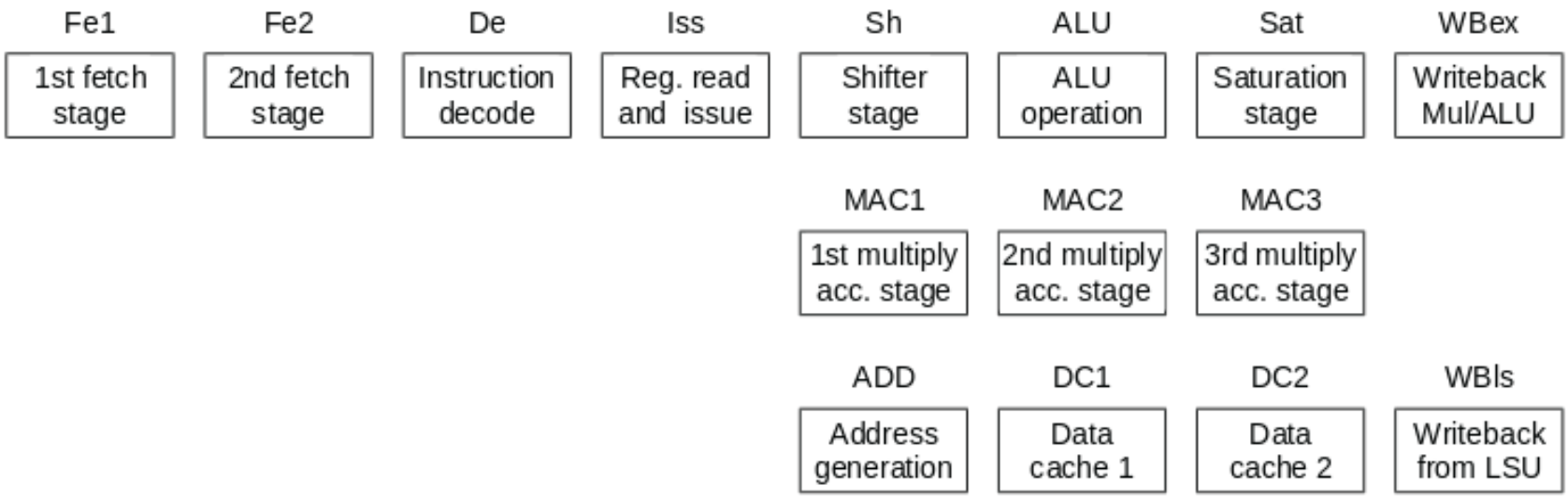
Performed at Basic Block Granularity.

Important to consider

- Pipeline Architecture of the target processor.
- 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.



Lets assume,

- Pipeline is empty at start of Basic Block.
- All data is available without latency.

Effects due to Data and Control Hazards

For each Basic Block from Binary Code,

- Parse instructions sequentially.
- Simulate progress in pipeline.
- Identify interlocking (hazards) and add penalty.
- Annotate to mapped block in Source Code.

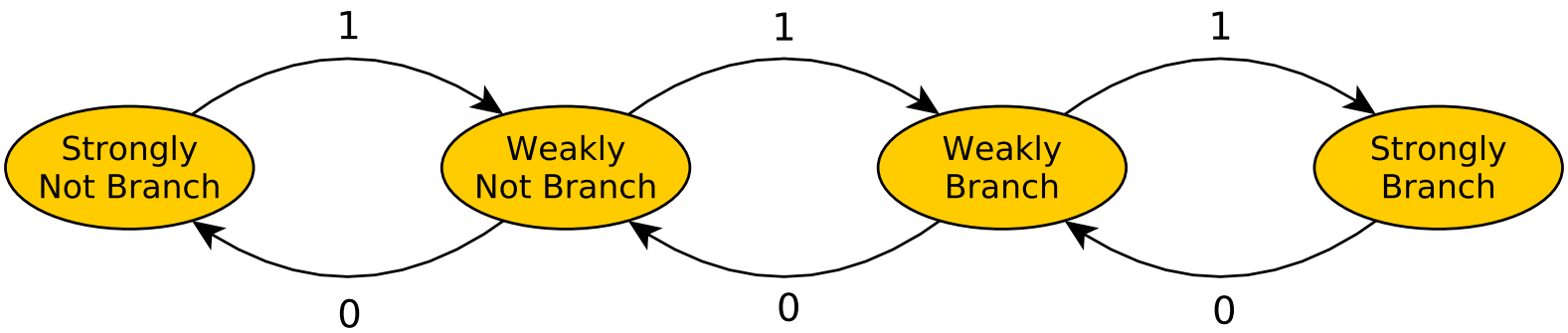
```
...
for(i<0; i<10; i++) {}
    execCycles += 23;
    sum += array[i];
    ...
}
...
```

Branch Prediction

- Reduces number of pipeline flushes.
- Major impact on performance.
- Branch Prediction Unit is simulated to account for this.

Branch Prediction Algorithm

- Simulates the BPU on ARM Cortex A5
- 125 entry Branch History Table.
- State Machine for each Entry. 2 bit state information.



Branch Prediction API

Branch Prediction Simulator 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

- Start and End address of Basic Block extracted from Binary.
- Annotation done at beginning of Basic Block.
- Depending on outcome, penalty is subtracted from `execCycles`.

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

Cache Simulator

Cache Hierarchy on target device.

	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

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

- Start address and size of Basic Blocks extracted from Binary.
- Annotation at beginning of mapped Basic Block in Source Code.
- Return value added to `memAccessCycles`.

```
1  ...
2  for(i=0; i<10; i++) {
3      execCyles += 23;
4      execCycles -= (branchPred_enter(0x348, 0x380) ? 7 : 0);
5      memAccessCycles += simICache(0x348, 56);
6      sum += array[i];
7      ...
8  }
9  ...
```

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);
```

Data Cache Simulation

Address from host cannot be used for simulation.

Will lead to inaccuracies, because

- Different sizes of Basic Data Types.
- Different Memory Alignment on Host and Target.

Memory Access Reconstruction

- Inspired by [TODO]
- Resolve address of each load/store instruction on target device.

Memory Access Reconstruction

- Resolve Physical Address of each variable in target memory.
- Identify variable being accessed by each load/store.
- Identify index information from source code.
- Simulate Data Cache.

Resolve Physical Address of Variables

Global Variables

- Bare-Metal Execution. Address fixed at compile time.
- Extracted by Static Analysis of binary.

Annotate address in code as follows,

```
1  unsigned int globVar[100];
2  unsigned long globVar_addr = 0x7c8;
```

Resolve Physical Address of Variables

Local Variable

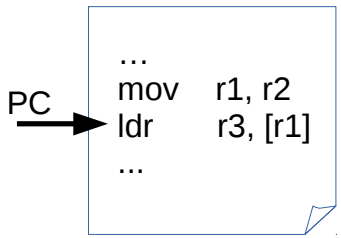
- Stored in Stack, address only known at run-time.
- Address relative to stack, can be known statically.
- Simulate growth of stack at run-time to resolve physical address.

```
1  unsigned long CSIM_SP = 0x1d4c8;          // Initial Value of SP.
2
3  int foo()
4  {
5      CSIM_SP += 88;                        // Size of Stack Frame for foo.
6
7      int localVar;
8      unsigned long localVar_addr = 0x8;    // Address relative to SP.
9      ...
10 }
```

Identify Load/Store Operations in Binary Code

To identify variable accessed by a load/store instruction,

- Binary code is functionally simulated.
- Register state is maintained, and updated as per instruction.
- Branch instructions are ignored.
- Address for each load/store is extracted.
- Variable being accessed is known.



Assembly Code

r0	r1	r2	...	r15
0	7c8	7c8	...	346

Register State

Annotation of Memory Access

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

- Each line is parsed using a custom C Parser.
- Index information is extracted.
- Annotation as follows.

```
1  ...
2  for (i=0; i < 100; i++) {
3      sum += globalVar[i];
4      memAccessCycles += simDCache ( globalVar_addr + i * 4, True );
5  }
6  localVar = sum / 100;
7  memAccessCycles += simDCache ( SP + localVar_addr, False );
8  ...
```

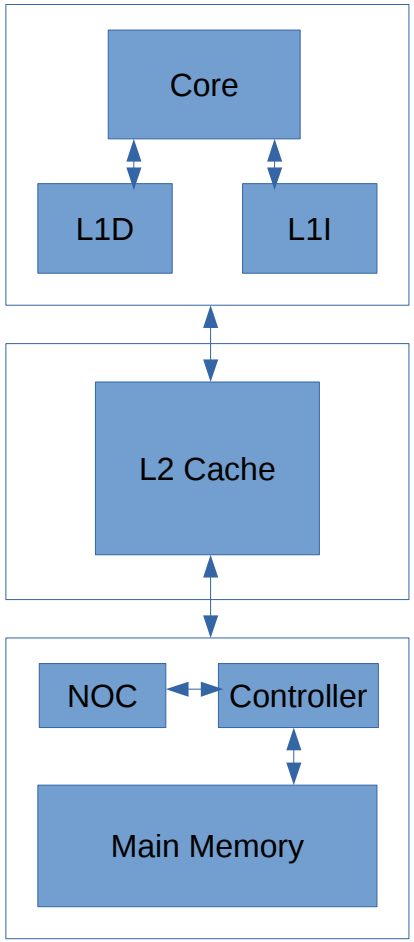
Timing Estimation

- Annotated Source Code is compiled and run on Host Machine.
- Time spent in Active state and Idle State of CPU is reported.
- Additionally, trace information from Cache is generated.

For estimating power, the power state model approach has been used.

- Power consumed by each component in active and idle state is known.
- Trace information for each basic block, shows components that were active and the duration.
- Total power being consumed at a time, can be estimated.

Components for Power Estimation



Target System is divided into components as follows

- Core and L1 Caches
- L2 Cache
- External Memory along with controller and NOC.

Test Setup

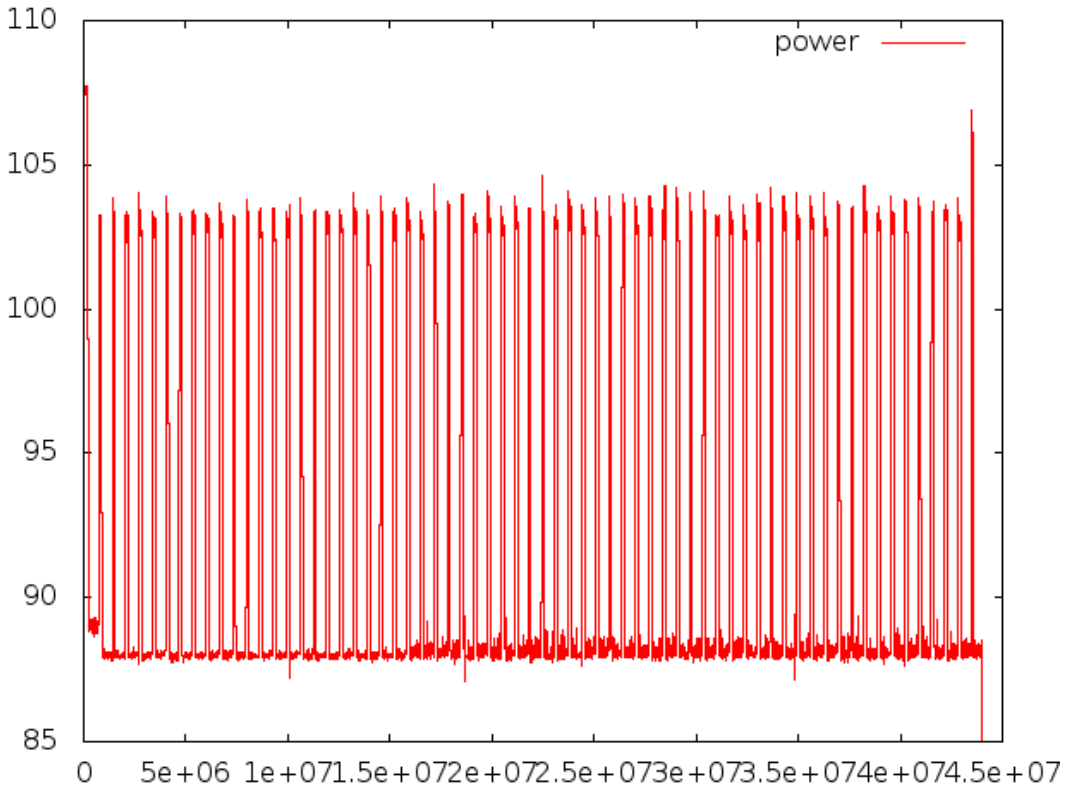
For testing, ARM Cortex A5 core was used. Results from ARM Performance Measurement Unit were compared.

- Number of Cache Misses.
- Total Cycles.

Accurate Prediction of Cache Misses, will verify the correctness of the instrumentation.

ADPCM Benchmark

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



Sieve Benchmark

	HCS	Actual	Accuracy
Total Cache Miss	933109	934250	99.88%
Total Cycles	99068687	100403731	98.67%