EC 535: Introduction to Embedded Systems

Today:

Metrics for Embedded Systems

Quantitative Analysis: Metrics and Performance

- Does the brake-by-wire software always activate the brake within 1 ms?

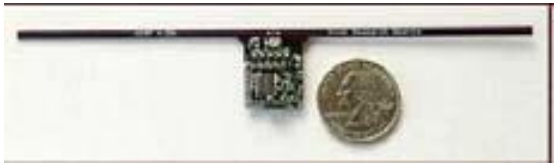
  Safety-critical embedded systems

- Can this app drain my phone battery in an hour?

  Consumer electronics

- How much energy does the sensor node need?

  Sensor nets, biomedical apps



https://ai-benchmark.com/

arXiv:1910.06663v1 [cs.PF] 15 Oct 2019

AI Benchmark: All About Deep Learning on Smartphones in 2019

Andrey Ignatov, ETH Zurich, andrey@vision.ee.ethz.ch
Radu Timofte, ETH Zurich, timofter@vision.ee.ethz.ch
Andrei Kulik, Google Research, akulik@google.com
Seungsoo Yang, Samsung, Inc., ss1.yang@samsung.com
Ke Wang, Huawei, Inc., michael.wangke@huawei.com
Felix Baum, Qualcomm, Inc., fbaum@qti.qualcomm.com
Max Wu, MediaTek, Inc., max.wu@mediatek.com
Lirong Xu, Unisoc, Inc., lirong.Xu@unisoc.com
Luc Van Gool*, ETH Zurich, vangool@vision.ee.ethz.ch

Abstract

The performance of mobile AI accelerators has been evolving rapidly in the past two years, nearly doubling with each new generation of SoCs. The current 4th generation of mobile NPUs is already approaching the results of CUDA-compatible Nvidia graphics cards presented not long ago, which together with the increased capabilities of mobile deep learning frameworks makes it possible to run complex and deep AI models on mobile devices. In this paper, we evaluate the performance and compare the results of all chipsets from Qualcomm, HiSilicon, Samsung, MediaTek and Unisoc that are providing hardware acceleration for AI inference. We also discuss the recent changes in the Android ML pipeline and provide an overview of the deployment of deep learning models on mobile devices. All numerical results provided in this paper can be found and are regularly updated on the official project website [1].

1. Introduction

Over the past years, deep learning and AI became one of the key trends in the mobile industry. This was a natural fit, as from the end of the 90s mobile devices were getting equipped with more and more software for intelligent data processing – face and eyes detection [20], eye tracking [53], voice recognition [51], barcode scanning [84], accelerometer-based gesture recognition [48, 57], predictive text recognition [74], handwritten text recognition [4], OCR [36], etc. At the beginning, all proposed methods were mainly based on manually designed features and very compact models as they were running at best on devices with a single-core 600 MHz Arm CPU and 8-128 MB of RAM. The situation changed after 2010, when mobile devices started to get multi-core processors, as well as powerful GPUs, DSPs and NPUs, well suitable for machine and deep learning tasks. At the same time, there was a fast development of the deep learning field, with numerous novel approaches and models that were achieving a fundamentally new level of performance for many practical tasks, such as image classification, photo and speech processing, neural language understanding, etc. Since then, the previously used hand-crafted solutions were gradually replaced by considerably more powerful and efficient deep learning techniques, bringing us to the current state of AI applications on smartphones.

Nowadays, various deep learning models can be found in nearly any mobile device. Among the most popular tasks are different computer vision problems like image classification [38, 82, 23], image enhancement [27, 28, 32, 30], image super-resolution [17, 42, 83], bokeh simulation [85], object tracking [87, 25], optical character recognition [56], face detection and recognition [44, 70], augmented reality [3, 16], etc. Another important group of tasks running on mobile devices is related to various NLP (Natural Language Processing) problems, such as natural language translation [80, 7], sentence completion [52, 24], sentence sentiment analysis [77, 72, 33], voice assistants [18] and interactive chatbots [71]. Additionally, many tasks deal with time series processing, *e.g.*, human activity recognition [39, 26], gesture recognition [60], sleep monitoring [69], adaptive power management [50, 47], music tracking [86] and classification [73]. Lots of machine and deep learning algorithms are also integrated directly into smartphones firmware and used as auxiliary methods for estimating various parameters and for intelligent data processing.

*We also thank Oli Gaymond (ogaymond@google.com), Google Inc., for writing and editing section 3.1 of this paper.

[1] http://ai-benchmark.com

1

AI Benchmark: All About Deep Learning on Smartphones in 2019

Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, Luc Van Gool

The performance of mobile AI accelerators has been evolving rapidly in the past two years, nearly doubling with each new generation of SoCs. The current 4th generation of mobile NPUs is already approaching the results of CUDA-compatible Nvidia graphics cards presented not long ago, which together with the increased capabilities of mobile deep learning frameworks makes it possible to run complex and deep AI models on mobile devices. In this paper, we evaluate the performance and compare the results of all chipsets from Qualcomm, HiSilicon, Samsung, MediaTek and Unisoc that are providing hardware acceleration for AI inference. We also discuss the recent changes in the Android ML pipeline and provide an overview of the deployment of deep learning models on mobile devices. All numerical results provided in this paper can be found and are regularly updated on the official project website.

Face Recognition

Image Classification

Image Enhancement...

Is your **smartphone** capable of running the latest **Deep Neural Networks** to perform these AI-based tasks? Is it fast enough?

View Detailed Results

Chart

Model	SoC	RAM	Year	Android	Updated	Lib	CPU-Q Score	CPU-F Score	INT8 CNNs	INT8 Transformer	INT8 Accuracy	FP16 CNNs	FP16 Transformer	FP16 Accuracy	INT16 CNNs	INT8 Parallel	FP16 Parallel	INT8 Memory	FP16 Memory	AI Score
Oppo Find X8 Pro	Dimensity 9400	16GB	2024	15	10.24	mm	160	157	815	2876	77.5	1062	1379	97.8	336	51	62	3100	2700	10319
Oppo Find X8	Dimensity 9400	16GB	2024	15	10.24	mm	162	158	795	2864	77.5	1040	1374	97.8	326	50	61	3100	2700	10225
vivo X200 Pro	Dimensity 9400	16GB	2024	15	10.24	mm	148	134	810	2823	77.5	1044	1349	97.8	335	56	61	3100	2800	10132
vivo X200	Dimensity 9400	16GB	2024	15	10.24	mm	148	134	809	2819	77.5	1045	1345	97.8	336	56	61	3100	2800	10122
vivo X200 Pro Mini	Dimensity 9400	16GB	2024	15	10.24	mm	145	133	807	2805	77.5	1041	1347	97.8	336	60	62	3100	2800	10095
vivo X100 Pro	Dimensity 9300	16GB	2023	14	10.24	mm	113	116	649	1974	76.4	863	957	97.8	276	43	53	3100	2800	7532
vivo X100	Dimensity 9300	16GB	2023	15	10.24	mm	114	116	633	1961	76.4	851	946	97.8	269	41	53	3100	2800	7446
Xiaomi 14T Pro	Dimensity 9300+	12GB	2024	14	10.24	mm	119	114	616	1934	76.4	829	930	97.8	258	48	53	3100	2700	7307
vivo X100s	Dimensity 9300+	12GB	2024	14	10.24	mm	104	110	619	1936	76.4	831	927	97.8	265	43	54	3100	2800	7306
Xiaomi Redmi K70 Ultra	Dimensity 9300+	16GB	2024	14	10.24	mm	124	117	608	1922	76.4	825	932	97.8	256	42	53	3100	2800	7295
vivo X100s Pro	Dimensity 9300+	16GB	2024	14	10.24	mm	107	115	608	1921	76.4	824	921	97.8	261	42	55	3100	2800	7251
Apple iPhone 16 Pro	Apple A18 Pro	8GB	2024	iOS 18.1	11.24	coreml	159	160	247	1499	100	743	916	100	0	17	50	3200	3200	5845
Samsung Galaxy S24 Ultra	Snapdragon 8 Gen 3	12GB	2024	14	10.24	qhqh	106	107	722	1607	69.9	619	389	95.3	48	90	128	2200	2100	5374
Samsung Galaxy S24	Snapdragon 8 Gen 3	12GB	2024	14	10.24	qhqh	92	90	722	1601	69.9	631	357	95.3	48	94	132	2200	2100	5295
Apple iPhone 15 Pro	Apple A17 Pro	8GB	2023	iOS 18	10.24	coreml	126	128	249	1288	100	766	807	100	0	16	49	3200	3200	5286
Asus ROG Phone 8	Snapdragon 8 Gen 3	16GB	2024	14	10.24	qhqh	105	110	720	1578	69.9	593	387	95.3	47	96	101	2300	2200	5278

Power Efficiency Ranking

Processor	AI Accelerator	Year	Lib	Inference Mode	INT8, FPS per Watt	FP16, FPS per Watt	Power Efficiency Score
Snapdragon 8 Gen 2	Hexagon DSP / HTP Gen 2	2022	qh.qh	FAST SINGLE ANSWER	48.9	11.1	23.3
Snapdragon 8 Gen 2	Hexagon DSP / HTP Gen 2	2022	qh.qh	SUSTAINED SPEED	44.8	11.6	22.8
Dimensity 9200	APU 690	2022	mm	SUSTAINED SPEED	24.8	9.2	15.1
Dimensity 9200	APU 690	2022	mm	FAST SINGLE ANSWER	26.9	8.2	14.9
Snapdragon 8 Gen 1	Hexagon DSP / HTP	2021	qh.qh	SUSTAINED SPEED	29.2	7.1	14.4
Dimensity 9000	APU 590	2021	mm	SUSTAINED SPEED	21.5	7.3	12.5
Dimensity 9000	APU 590	2021	mm	FAST SINGLE ANSWER	22	6.3	11.8
Snapdragon 888	DSP (Hexagon 780) + GPU (Adreno 660)	2020	qh.qg	SUSTAINED SPEED	50	2.8	11.8
Dimensity 800	APU 3.0 (4 cores)	2020	nn	SUSTAINED SPEED	15.8	5.2	9.1
Google Tensor G2	Google Tensor TPU 2.0	2022	nn	SUSTAINED SPEED	12.6	5.1	8.0
Dimensity 820	APU 3.0 (4 cores)	2020	nn	SUSTAINED SPEED	14.7	4.4	8.0
Dimensity 1000+	APU 3.0 (6 cores)	2019	nn	SUSTAINED SPEED	12.1	4.7	7.5
Google Tensor	Google Tensor TPU	2021	nn	SUSTAINED SPEED	7	4.3	5.5
Snapdragon 865	DSP (Hexagon 698) + GPU (Adreno 650)	2019	hg	SUSTAINED SPEED	6.1	2.6	4.0

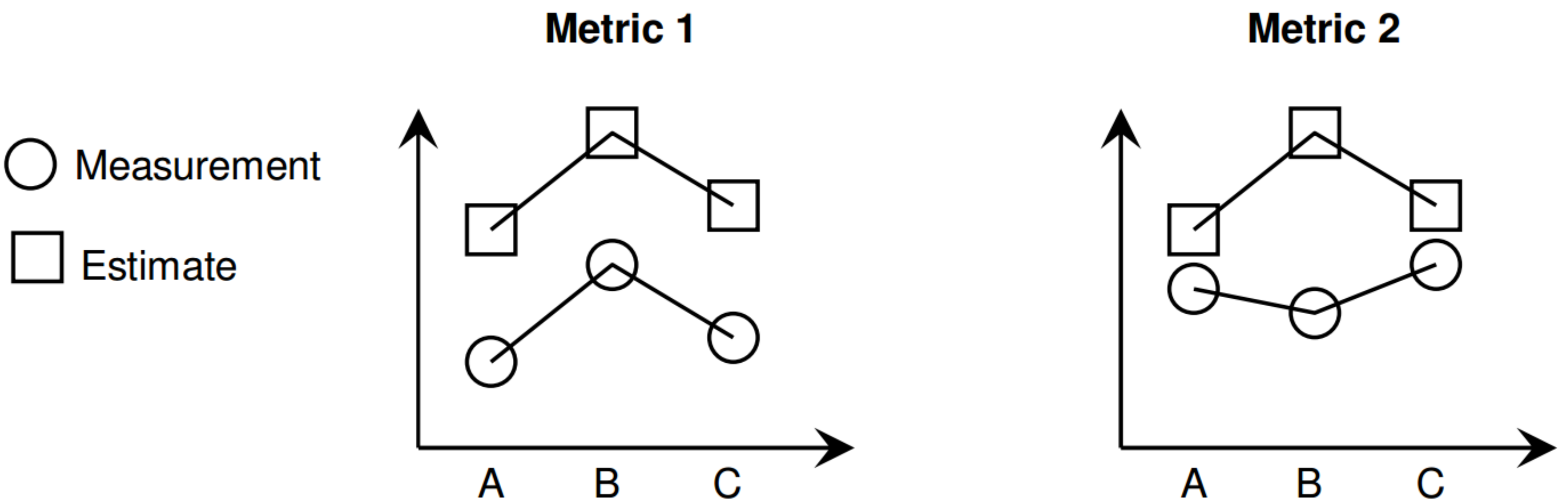
Metrics of Performance and Cost

- Design quality factor that cannot be measured until the design is complete
  - Cost – investment *into* a design
  - Performance – investment *from* a design

- Selecting a good metric is not as easy as it sounds
  - How to measure processor performance?

- Metric properties
  - Target dependency
  - Accuracy
  - Fidelity

Accuracy vs. Fidelity



- Accuracy $=1-\frac{|E-M|}{M}$

  E: estimate M: measurement

  - Estimate: 9 cycles
  - Measured: 10 cycles
  - Accuracy: 0.9

- Fidelity
  - How well a metric works over different designs
  - Checks consistency among each pair

Metric Classification

Algorithmic	O(n) (*Big-O complexity*)
Technology-dependent	Estimation of Power consumption Relative power estimation (SW, HW) Area and Cycle Time
Architecture-dependent (SW)	Instruction-accurate profiling Cycle-accurate profiling Cycle-accurate instruction-traces Static memory requirements
Architecture-dependent (HW)	Static analysis of FSMD source code Profiling of FSMD operations
Reference	NCLOC Cyclomatic Complexity Profiling of C-code operations Profiling of C-code memory accesses

Algorithmic Metrics

- Big-O complexity
  - Especially useful for large problems
  - Ignores constants and multipliers
    - O(1000*n + 10000) and O(1*n + 1) are both O(n)

- Usually not enough for embedded systems
  - Resources are limited. Ignored constants may be crucial.
  - Need to use information on the hardware

Timing is Central to Embedded Systems

- Several timing analysis problems:
  - Worst-case execution time (WCET) estimation
  - Estimating distribution of execution times
  - Threshold property: can you produce a test case that causes a program to violate its deadline?
  - Software-in-the-loop simulation: predict execution time of particular program path

Basic problem: Execution time analysis of programs

Metrics for Real-time Embedded Systems

- Timing/timeliness
  - Average-case execution time vs. worst-case execution time vs. best-case execution time
  - Why WCET is important? How to estimate ACET/WCET/BCET? What are the factors that influence execution time?
- Resource usage/bottlenecks
  - Which part of my program is taking the most time: profiling
  - Memory usage, cache performance, etc.

Worst-Case Execution Time (WCET) & BCET

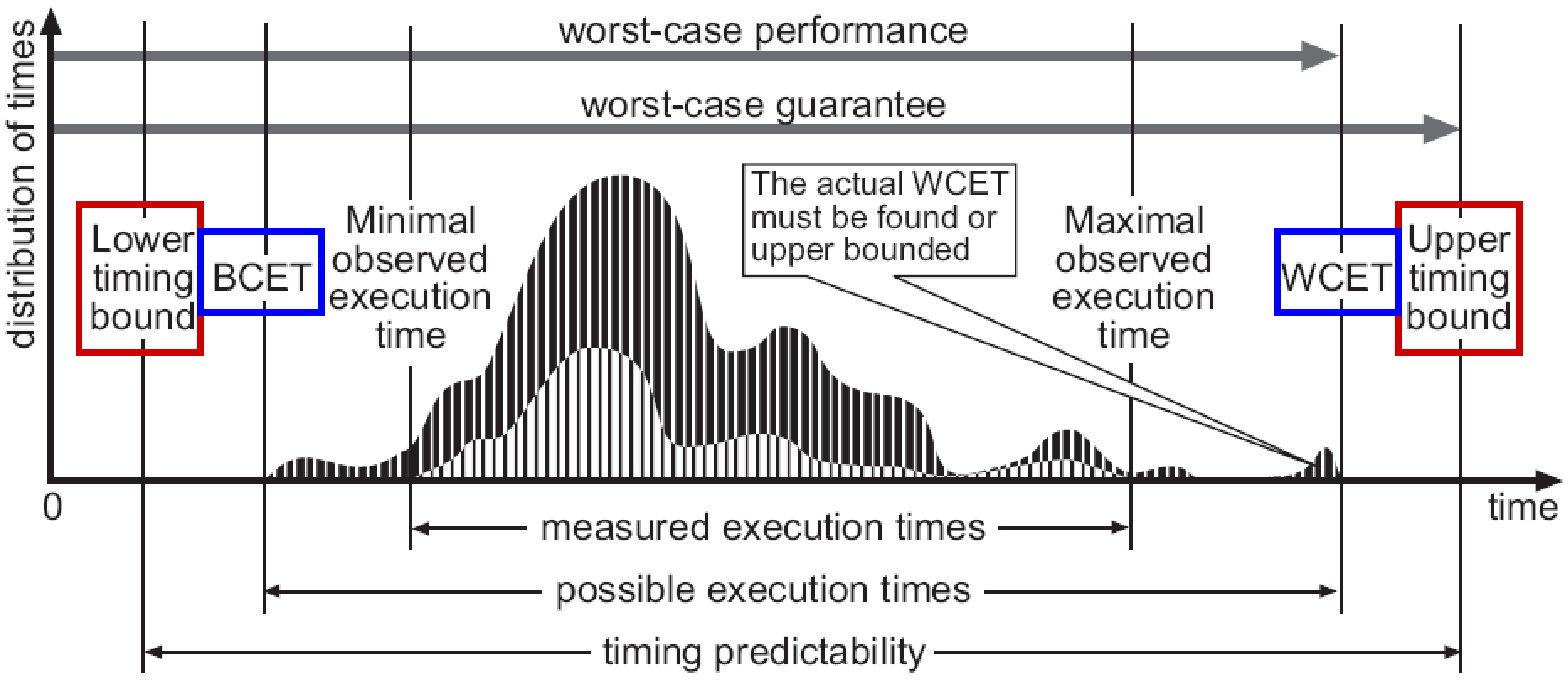


Figure from R.Wilhelm et al., ACM Trans. Embed. Comput. Sys, 2007.

WCET in Real-time Systems

- The WCET is central in the design of RT Systems:
  - Needed for Correctness (does the task finish in time?) and
  - Performance (find optimal schedule for tasks)

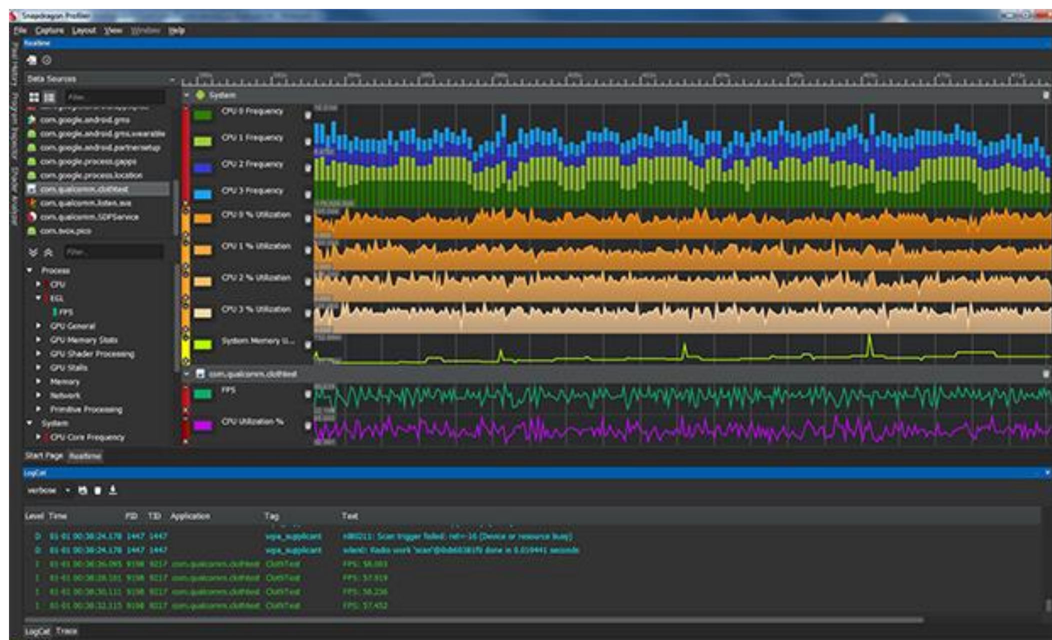
- The WCET problem:
  - Given the code for a software task AND the platform (OS + hardware) that it will run on,
  - Determine the WCET of the task.

- Estimating WCET is difficult (in general undecidable):
  - Embedded system assumptions: loops with finite bounds, no recursion, single threaded

Program Profiler

- Analyze the runtime behavior of a program
  - Which parts (functions, statements, . . .) of a program take how long?
  - How often are the functions called?
  - Which functions call which?
  - Memory consumption: memory accesses, memory leaks, cache performance

Profiling tools: perf, Valgrind, Gprof



Example - 1

- Traversing a 2-D array

```
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
       sq_mat[i][j] = 0;
    }
}
```

```
for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
       sq_mat[j][i] = 0;
    }
}
```

Try this: which one is faster? why are they different?

Example - 2

- Reading a file

```
//process each line
while(read = getline(…)) != -
1){
  process_line(read);
}
```

What if the file cannot fit in the cache or memory?

```
//get file length
fseek(infile, 0, SEEK_END);
fileLength = ftell(infile);
fseek(infile, 0, SEEK_SET);

//allocate buffer for the file
buf =
(char*)malloc(fileLength);

//read file
fread(buf, 1, fileLength,
infile);

//process each line
line = strtok(buf, "\n\r");
while (line != NULL){
  process_line(line);
  line = strtok(NULL, "\n\r");
}
```

Metric Classification

Algorithmic	O(n) (*Big-O complexity*)
Technology-dependent	Estimation of Power consumption Relative power estimation (SW, HW) Area and Cycle Time
Architecture-dependent (SW)	Instruction-accurate profiling Cycle-accurate profiling Cycle-accurate instruction-traces Static memory requirements
Architecture-dependent (HW)	Static analysis of FSMD source code Profiling of FSMD operations
Reference	NCLOC Cyclomatic Complexity Profiling of C-code operations Profiling of C-code memory accesses

Architecture-dependent Metrics

- Number of instructions/cycles, memory requirement, etc.

- More accurate, but also more expensive
  - May need to simulate or to use the hardware

- Useful to optimize a software for a target hardware

- Potential problems:
  - Simulation model may introduce some abstraction
    - e.g., a cycle-accurate simulator may not be able to model the processor bus and other peripherals
  - Simulated hardware may be different from the target

Instruction-accurate profiling

- SimIt-ARM instruction-accurate profiling:

```
$ ema -f nwfpe.bin small
The result is 4900
Total user time : 0.070 sec.
Total system time: 0.001 sec.
Simulation speed : 3.088e+07 inst/sec.
Total instructions : 2191774 (2M) including 7211 nullified
Total 4K memory pages allocated : 370
```

- Execution Time = Total Instructions / Simulation Speed

Cycle-accurate profiling

```
$ sima -f nwfpe.bin small
The result is 4900
Total icache reads: 2551776
Total icache read misses: 457
icache hit ratio: 99.982%
Total itlb reads: 2551819
Total itlb read misses: 25
itlb hit ratio: 99.999%
Total dcache writes: 369657
Total dcache write misses: 3654
Total dcache reads: 871393
Total dcache read misses: 207
dcache hit ratio: 99.689%
Total dtlb reads: 1241050
Total dtlb read misses: 26
dtlb hit ratio: 99.998%
Total biu accesses: 4315
biu activity: 3.438%
Total allocated OSMs : 2551819
Total retired OSMs : 2551817
Total cycles : 3131710
Equivalent time on 206.4MHz host: 0.0152 sec.
```

Static memory requirements

- Code and data space requirements using *size*:

```
$ arm-linux-size small
  text    data     bss     dec     hex filename
362479    4172    5140  371791   5ac4f small
```

- text: Code size
- data: Initialized data size
- bss: Non-initialized data size
- dec: Total bytes required in decimal
- hex: Total bytes required in hex

Simulation Speed

- Common terms
  - Target — The processor to be simulated
  - Host — The workstation that runs the simulator
- Slow down factor: a rough but common metric
  - Host speed/simulation speed
  - The number of host instructions to interpret one target instruction
  - If the simulator executes 100K Inst/sec on a 1GHz host (1 billion cycles/sec)
  - Then roughly slow down = 1G/100k = 10,000

Processor Simulators

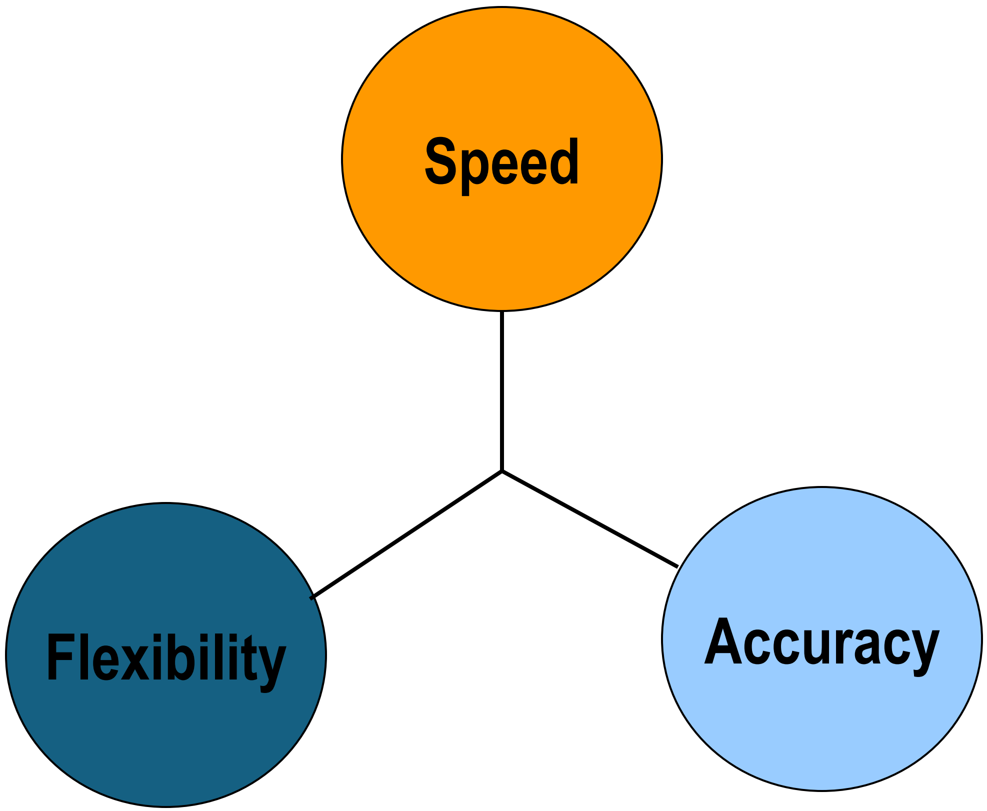
- Instruction set simulator (ISS)
  - Interpreter
    - The common approach, flexible, reasonably fast
    - Slowdown ~ 100
  - Static compiled simulator
    - The less common approach, not as flexible, very fast
    - Slowdown ~ 1-10
  - Dynamic compiled simulator
    - The high-tech approach, flexible, very fast
    - Difficult to develop and to port (similar to writing a VM)
    - Slowdown ~ 1-10

Using ISSs

- Profiling benchmarks
  - Count the number of instructions executed
  - Find the kernel loops (the ones that execute most frequently)
- Execution trace generation
  - Get the memory footprint
  - Can plug in a cache model to estimate cache performance
  - Can plug in a branch predictor model to estimate branch prediction performance
- Cross-development of software
  - Evaluating functional correctness
  - Debugging : reversible execution (backtrack to bug source)
    - E.g., Simics (originally from Virtutech, then Intel / Wind River Systems)

Concerns

Faster is better.



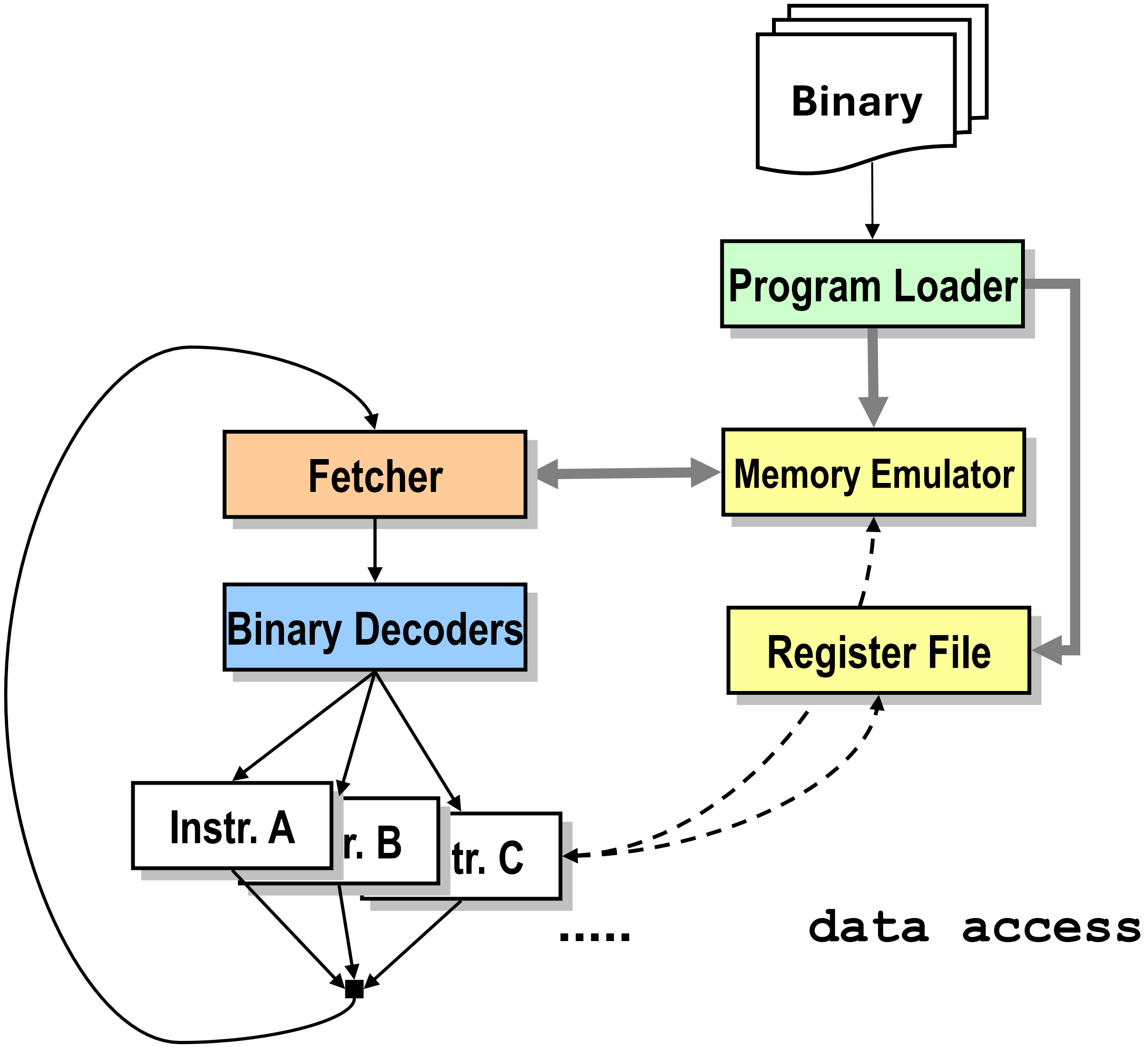
Easy to extend is important.

Accuracy increases confidence.

Interpreter Components

- Program loader
  - Reads the binary program, parse the headers and identify code and data sections
  - Initializes PC, stack pointer, etc.
- Memory emulator
  - Simplest case: an array to emulate the memory of the target system, works if the target address space is small
  - More sophisticated: a page table
    - In SimIt-ARM, 4k pages are allocated on demand

ISS — Interpreter



Interpreter Components (cont'd)

- Fetcher
  - Usually a single line of C code to read the memory
- Binary decoder
  - Map instruction words to actual individual interpretation routines
  - Complexity varies depending on the ISA encoding

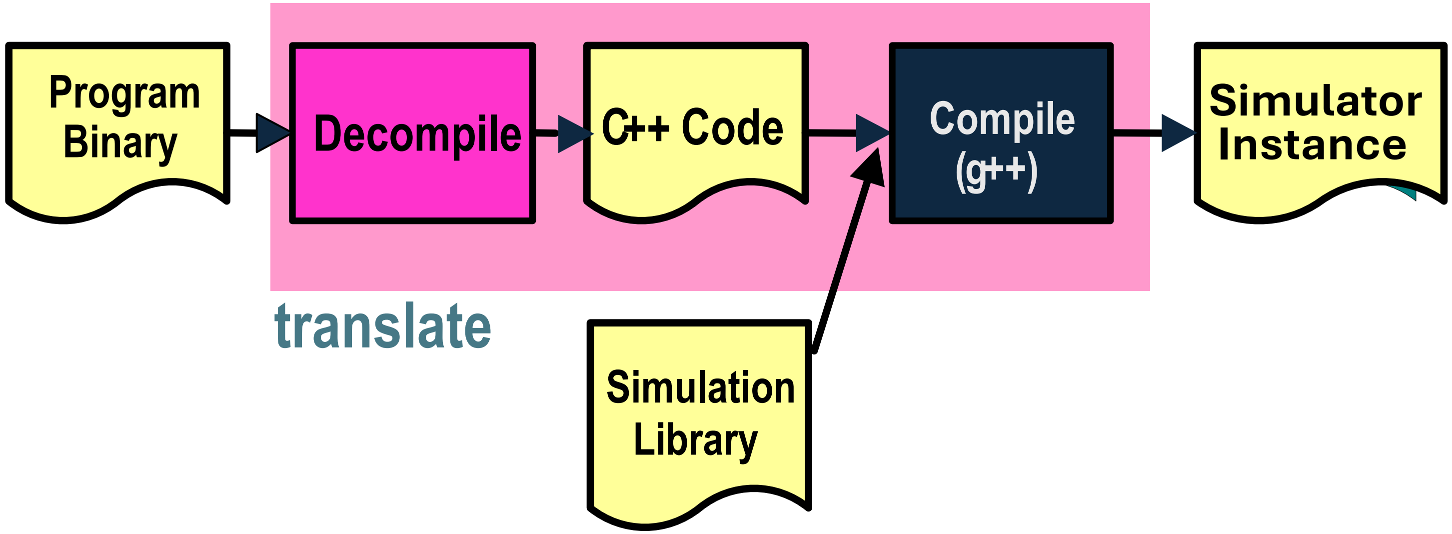
A simple decoder

opcode	operands

```
opcode = instruction_word >> 25;
switch (opcode) {
  case add:
interpret_add(instruction_word);
          break;
  case sub:
interpret_sub(instruction_word);
          break;
  ......
}
```

Static-Compiled ISS

- Static-compiled
  - Remove fetch/decode overhead
  - Translate the target binary to host binary
  - Execute the host binary
- Can do so via C/C++
  - Advantage: Portability



Static-compiled ISS

- Compared to interpretation
  - + No fetching/decoding overhead
  - - Need to compile for every benchmark
  - - Compilation can take long due to bloated C code
  - - Cannot simulate self-modifying code, e.g. OS

Dynamic-compiled ISS

- Static-compiled ISS cannot simulate self-modifying code, e.g., OS
- Static-compiled ISS is complex to use
  - Need to invoke gcc for each binary to simulate, slow
- Dynamic-compiled ISS uses similar idea, but generates host binary in run time
  - Similar speed to static-compiled, but free of its problems
  - Hard to implement, dependent on host OS
  - JVM is in this category
  - Others: Shade (from SUN), Embra (Stanford)
- A lot of work involved to write one

Metric Classification

Algorithmic	O(n) (*Big-O complexity*)
Technology-dependent	Estimation of Power consumption Relative power estimation (SW, HW) Area and Cycle Time
Architecture-dependent (SW)	Instruction-accurate profiling Cycle-accurate profiling Cycle-accurate instruction-traces Static memory requirements
Architecture-dependent (HW)	Static analysis of FSMD source code Profiling of FSMD operations
Reference	NCLOC Cyclomatic Complexity Profiling of C-code operations Profiling of C-code memory accesses

Power estimation

- Power consumption of a processor
  - Voltage & frequency
  - Instructions per second
  - Accelerator usage (GPU, floating point unit, etc.)
  - Cache hit/miss per second
  - Memory access per second
  - Branch prediction hit/miss per second
  - CPU utilization
  - Bus utilization
  - ...
- No certain answer on what to use
- But we can still make use of the “relative” metrics
  - Which program consumes more memory power?
  - Statistical techniques

Reference Metrics

- Higher abstraction when only a reference implementation is available

- NCLOC
  - Non-Comment Number Of Lines

- Cyclomatic Complexity
  - Analysis on control dependency graph

Profiling C Code

- Profiling:
  - Invasive: modify the program, i.e., code instrumentation
  - Non-invasive: statistic sampling of the program
- Profiles:
  - Flat profile
  - Call graph
  - Annotated sources
- Tools:
  - gprof
  - gcov
  - valgrind
  - oprofile

Profiling C code

- small.c

```
#include "stdio.h"
int two(int limit) {
  int a, i;
  a = 0;
  for (i=0; i<limit; i++)
    a += i;
}

int one(int limit) {
  int i, a[50];

  for (i=0; i<limit; i++)
    a[i % 50] = i + two(i);
  return a[49];
}

int main() {
  int j, a;
  a = 0;
  for (j=0; j<1000; j++)
    a = a + one(j);
  printf("The result is %d\n", a);
  return 0;
}
```

gprof for profiling C code

- Enable profiling during compilation

```
> gcc -g -p small.c
```

- When run, the binary will create a file "gmon.out", which can be analyzed by gprof

```
> ./a.out
> gprof a.out
```

- Some of the output looks like:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %    cumulative    self              self     total
 time    seconds    seconds    calls  us/call  us/call  name
 98.61      0.35      0.35   499500     0.71     0.71  two
  1.39      0.36      0.01     1000     5.00   360.00  one
```

gprof for profiling C code

- two() is much more significant
- However, two() is called by one() unnecessarily 98% of the time

In-class exercise:

- Run gprof with small.c
- Optimize something outside the main function to speed up the program
- Run gprof again, observe the change in the flat profile

- Submit a zip file on GradeScope (inclass exercise 3) including:
  - New code in a file named small_new.c
    - Write names/usernames of people in the team as comments
  - The old and the new flat profile

```c
#include "stdio.h"
int two(int limit) {
  int a, i;
  a = 0;
  for (i=0; i<limit; i++)
    a += i;
}

int one(int limit) {
  int i, a[50];

  for (i=0; i<limit; i++)
    a[i % 50] = i + two(i);
  return a[49];
}

int main() {
  int j, a;
  a = 0;
  for (j=0; j<1000; j++)
    a = a + one(j);
  printf("The result is %d\n", a);
  return 0;
}
```