Computer Architecture

计 算 机 体 系 结 构

第2讲：量化设计分析（2）

张献伟

xianweiz.github.io

DCS3013, 9/14/2022

Review Questions

- List some goals of architecture designs?

  Functional, high performance, reliable, low cost, low power, …

- Memory wall?

  Processor improves much faster than memory/disk.

- Tradeoffs in simulation?

  Speed, flexibility, accuracy.

- What MTTF and MTBF are evaluated for?

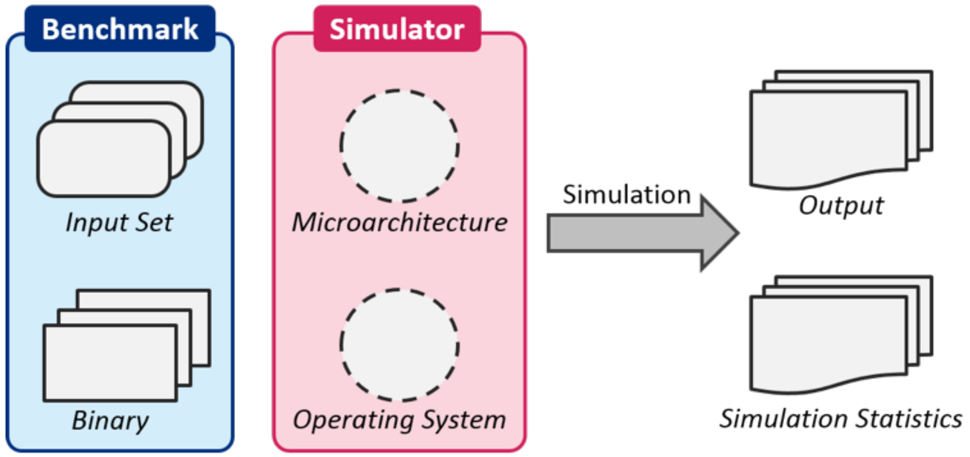
  Dependability/reliability/availability.

- Ways to measure performance?

  Direct/profiling, simulation, modeling.

Simulator[模拟器]

- What is an architecture (or architectural) simulator?
  - A tool that reproduces the behavior of a computing device
- Why use a simulator?
  - Leverage faster, more flexible software development cycle
  - Permits more design space exploration
  - Facilitates validation before hardware becomes available
  - Possible to increase/improve system instrumentation



Tradeoffs in Simulation[平衡]

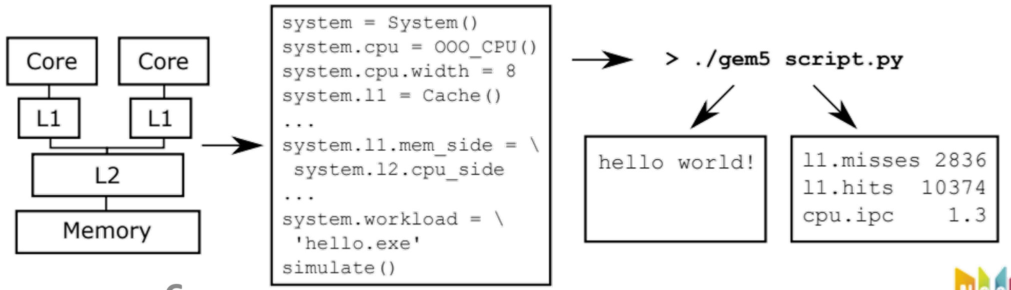
- Three metrics to evaluate a simulator
  - Speed, Flexibility, Accuracy
- **Speed**[速度]: How fast the simulator runs (xIPS, xCPS, slowdown)
- **Flexibility**[灵活性]: How quickly one can modify the simulator to evaluate different algorithms and design choices?
- **Accuracy**[准确度]: How accurate the performance (energy) numbers the simulator generates are vs. a real design (Simulation error)
- The relative importance of these metrics varies depending on where you are in the design process (what your goal is)

High-level Simulation[高层级模拟]

- Key Idea: Raise the abstraction level of modeling to give up some accuracy to enable speed & flexibility (and quick simulator design)
  - Get first-hand insights
- Advantages
  - Can still make the right tradeoffs, and can do it quickly
  - All you need is modeling the key high-level factors, you can omit corner case conditions
  - All you need is to get the “relative trends” accurately, not exact performance numbers
- Disadvantages
  - Opens up the possibility of potentially wrong decisions
  - How do you ensure you get the “relative trends” accurately?

Example Simulator: gem5

- gem5 = Wisconsin GEMS + Michigan m5
  - The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture.
  - Widely used in academia and industry
- Why gem5?
  - Runs real workloads
  - Comprehensive model library (memory, IO, Full OS, Web, …)
  - Rapid early prototyping (quickly test system-level ideas)
  - Can be wired to custom models (add detail where it matters, when it matters)



```
system = System()
system.cpu = OOO_CPU()
system.cpu.width = 8
system.l1 = Cache()
...
system.l1.mem_side = \
 system.l2.cpu_side
...
system.workload = \
 'hello.exe'
simulate()
```

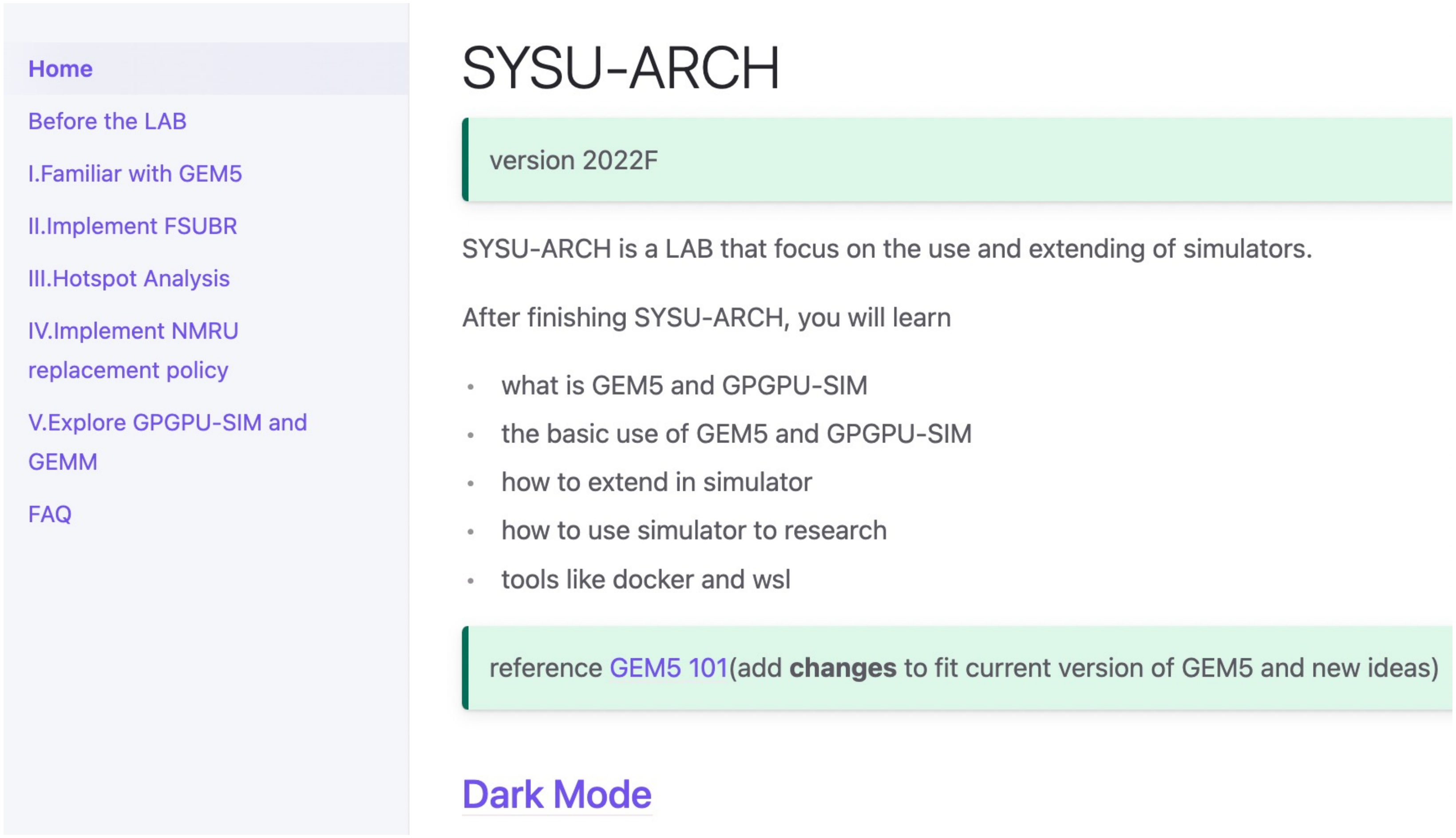
```
> ./gem5 script.py
```

```
hello world!
```

```
l1.misses 2836
l1.hits  10374
cpu.ipc     1.3
```

https://pages.cs.wisc.edu/~sinclair/courses/cs752/fall2020/handouts/lecture/archSim.pdf

Example Simulator: gem5 (cont.)



Home

Before the LAB

I.Familiar with GEM5

II.Implement FSUBR

III.Hotspot Analysis

IV.Implement NMRU replacement policy

V.Explore GPGPU-SIM and GEMM

FAQ

SYSU-ARCH

version 2022F

SYSU-ARCH is a LAB that focus on the use and extending of simulators.

After finishing SYSU-ARCH, you will learn

- what is GEM5 and GPGPU-SIM
- the basic use of GEM5 and GPGPU-SIM
- how to extend in simulator
- how to use simulator to research
- tools like docker and wsl

reference GEM5 101(add **changes** to fit current version of GEM5 and new ideas)

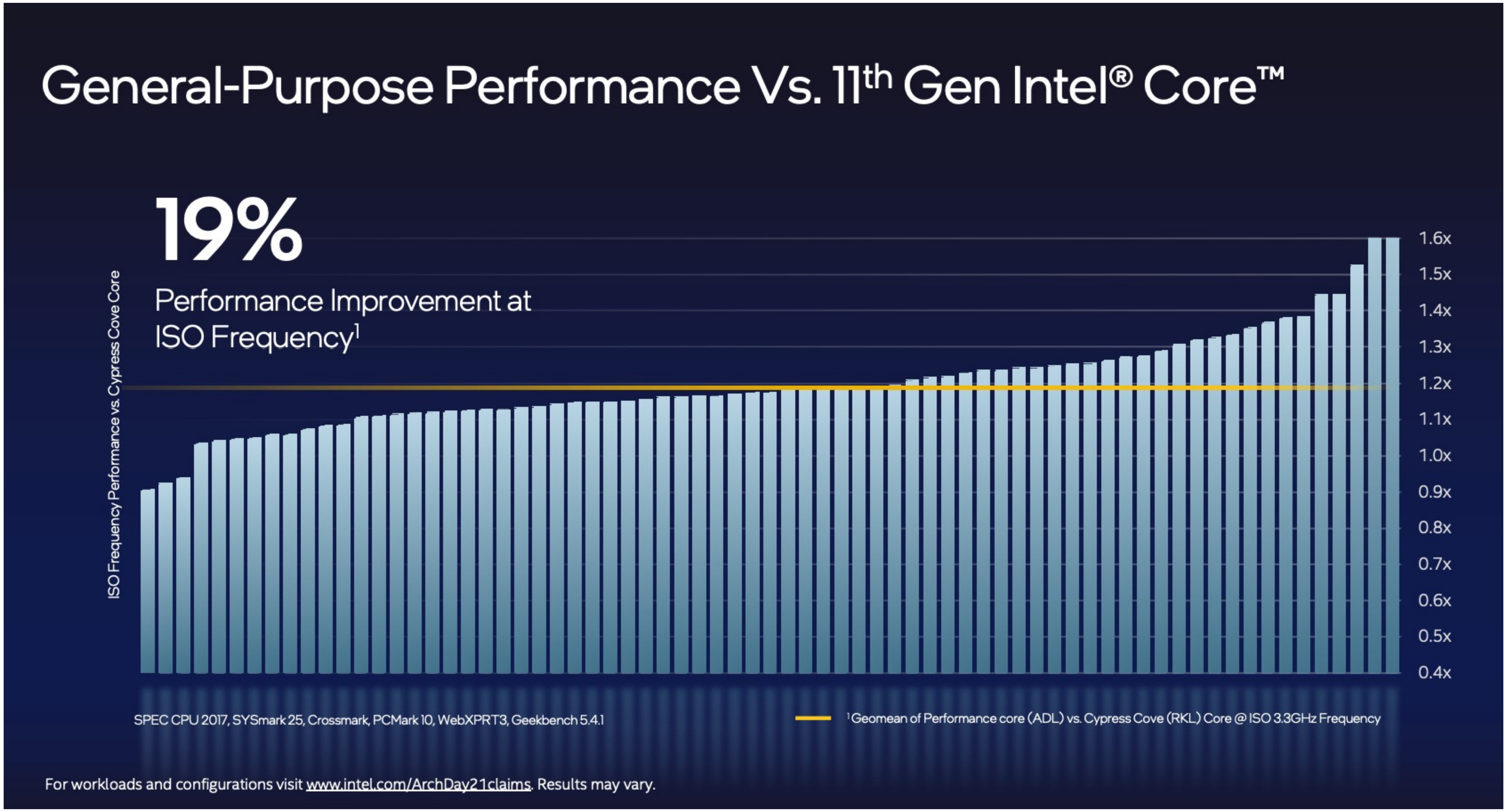
Dark Mode

Benchmarks[基准测试集]

- SPEC: Standard Performance Evaluation Corporation
- PARSEC: Princeton Application Repository for Shared Memory Computers
- Rodinia: GPGPU applications
- HPL: a High-Performance Linpack benchmark implementation
- MLPerf: a suite of performance benchmarks that cover a range of leading AI workloads widely in use
- MediaBench: Multimedia and embedded applications
- Transaction processing: TPC-C, SPECjbb
- EEMBC: embedded microprocessor benchmark consortium

Benchmarks (cont.)

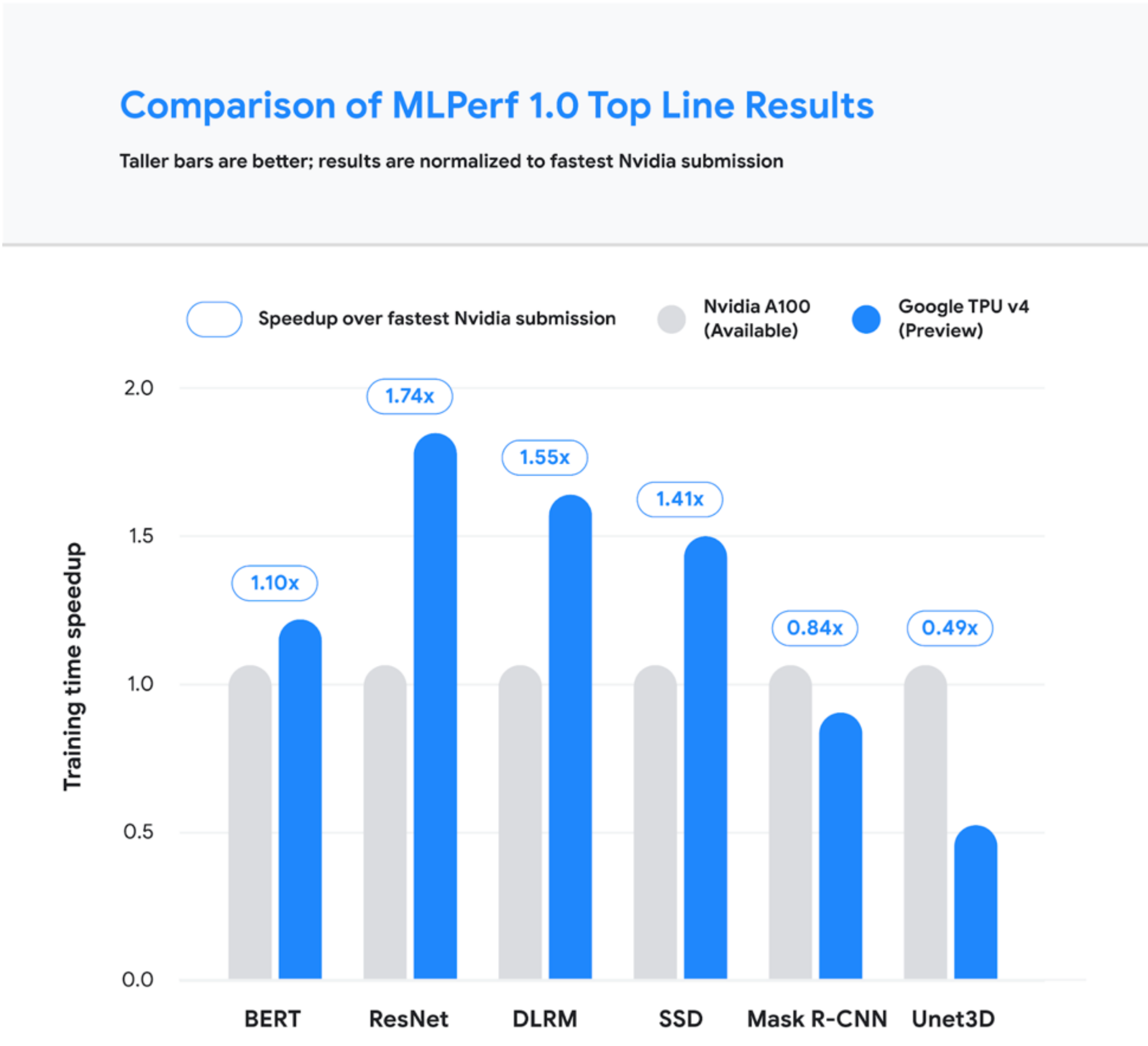
- Example: performance of Intel's newest CPU (2021)



https://download.intel.com/newsroom/2021/client-computing/intel-architecture-day-2021-presentation.pdf

Benchmarks (cont.)

- MLPerf
  - A broad ML benchmark suite for measuring performance of ML software frameworks, ML hardware accelerators, and ML cloud platforms



How to Summarize Performance

- Arithmetic mean (weighted arithmetic mean)[算术平均]
  - Considering the frequencies of programs in the workload
  - E.g., tracks execution time: $\sum_{i=1}^{n} \frac{T_i}{n}$ or $\sum_{i=1}^{n} W_i * T_i$
- Harmonic mean (weighted harmonic mean) of rates[调和平均]
  - E.g., track MFLOPS: $\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}}$
- Normalized execution time is handy for scaling performance (e.g., X times faster than Pentium 4)
- Geometric mean ==>$\sqrt[n]{\prod_{i=1}^{n} \text{execution\_}ratio_i}$ [几何平均]
  - The execution ratio is relative to a reference machine
    - Based on relative performance to a reference machine

Performance Evaluation[性能评估]

- Execution time and power are the main measure of computer performance
- Good products created when we have
  - Good benchmarks
    - For better or worse, benchmarks shape a field
  - Good ways to summarize performance
    - Reproducibility is important (should provide details of experiments)
- Given that sales is a function, in part, of performance relative to competition, companies invest in improving performance summary
  - If benchmarks/summary are inadequate, then choose between improving product for real programs vs. improving product to get more sales ===> Sales almost always wins!

Quantitative Principles (§1.8)[量化原则]

- Guidelines and principles that are useful in the design and analysis of computers
- Take advantage of parallelism[并行]
  - System level: multiple processors, multiple disks
  - Individual processor: instruction parallelism, e.g., pipelining
  - Detailed digital design: cache, memory
- Principle of locality[局部性]
  - Programs tend to reuse data and insts they have used recently
    - A program spends 90% of its execution time in only 10% of the code
- Focus on the common case[一般情况]
  - To make a trade-off, favor the frequent case over infrequent

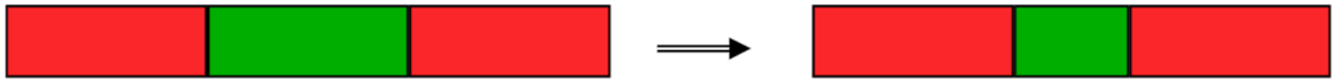
Amdahl‘s Law[阿姆达尔定律]

- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used
- 系统中对某一部件采用更快执行方式所能获得的系统性能改进程度，取决于这种执行方式被使用的频率，或所占总执行时间的比例
- Amdahl's law defines the speedup that can be gained by using a particular feature
  - Speedup due to some enhancement *E*:

$$Speedup_{overall} = \frac{ExTime_{withoutE}}{ExTime_{withE}} = \frac{Performance_{E}}{Performance_{withoutE}}$$

Amdahl's Law (cont.)

- Suppose that enhancement *E* accelerates a fraction of the task by a factor *S*, and the remainder of the task is unaffected



$$ExTime_{withE}$$

$$= ExTime_{withoutE} * [(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{S}]$$

$$Speedup = \frac{ExTime_{withoutE}}{ExTime_{withE}}$$

$$= \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{S}}$$

Amdahl's Law (cont.)

- Example 1: Floating point instructions can be improved to run 2X; but only 10% of actual instructions are FP. What is the overall speedup?
  - Fraction_E = 10%, S = 2, Speedup = 1/(90% + 10%/2) = 1.05

- Example 2: Assume we need to improve the performance of a graphics engine (assume 20% inst are FP Square root, 50% for all FP inst). Which choice is better?
  - Choice one: Speed up FP Square root by 10x

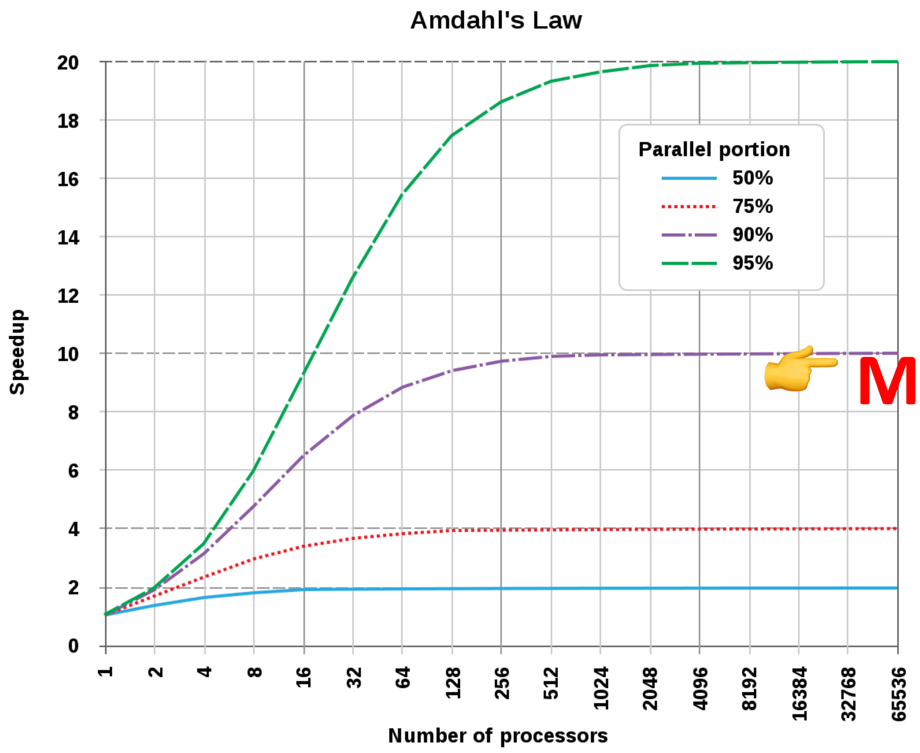
    1/(80% + 20%/10) = 1.22
  - Choice two: Speed up all FP instruction by 1.6x

    1/(50% + 50%/1.6) = 1.23

👉 **Focus on the common case !**

Amdahl's Law (cont.)

- A program’s speedup is limited by its serial part
  - For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be **20x**



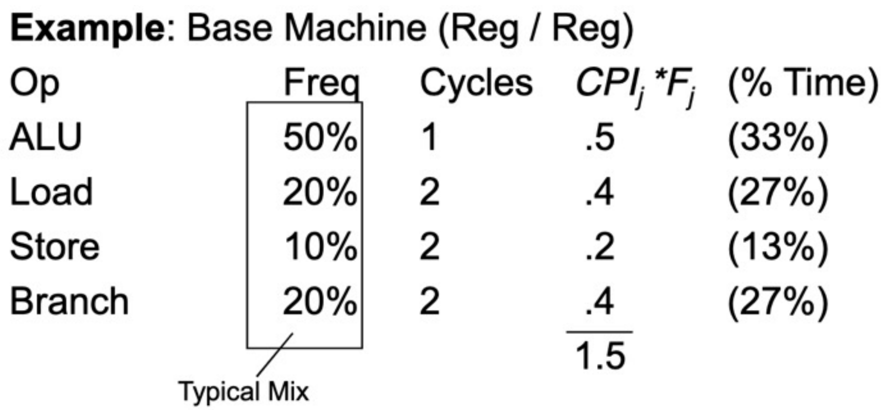
👉 **Make the fast case common !**

Computing CPU time

- CPU @ 2.5GHz
  - 2.5G ticks per second → 1/2.5G s/tick = 0.4ns / tick
  - Tick == clock == clock cycle
- CPU time for a program, i.e., #clock cycles to execute
  - CPU time = CPU clock cycles for a program x Clock cycle time
  - CPU time = CPU clock cycles for a program / Clock rate
- Clock cycles per instruction (CPI)
  - CPI = CPU clock cycles for a program / Instruction count
  - Reverse of IPC (instructions per cycle)
- CPU time = Inst count x CPI x Clock cycle time
  - $\frac{Instructions}{Program} x \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ cycle} = \frac{Seconds}{Program}$

Computing CPU time (cont.)

- $Average\ Cycles\ per\ Instruction\ (CPI) = \sum_{j=1}^{n} CPI_j * F_j$
  - Where CPI_j is the number of cycles needed to execute instructions of type *j*
  - and F_j is the percentage (fraction) of instructions that are of type *j*



Example: Base Machine (Reg / Reg)

Op	Freq	Cycles	$CPI_j * F_j$	(% Time)
ALU	50%	1	.5	(33%)
Load	20%	2	.4	(27%)
Store	10%	2	.2	(13%)
Branch	20%	2	.4	(27%)
			1.5	

Typical Mix

- $CPU\ time = Cycle\ time * \sum_{j=1}^{n} CPI_j * I_j$
  - I_j is the number of instructions of type *j*, and Cycle time is the inverse of the clock rate.

Computing CPU time (cont.)

- *CPI* is a function of the machine and program.
  - The *CPI* depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher *CPI* than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- It is common to each instruction took one cycle, making *CPI* = 1.
  - The *CPI* can be >1 due to memory stalls and slow instructions.
  - The *CPI* can be <1 on machines that execute more than 1 instruction per cycle (superscalar).

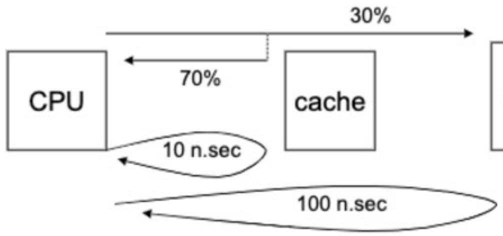
Aspects of CPU Performance

- $CPU\ time = \frac{Seconds}{program} = \frac{Instructions}{program} * \frac{Cycles}{Instructions} * \frac{Seconds}{Cycles}$

	Inst Count	CPI	Clock Rate
Program	❍		
Compiler	❍	❍	
Inst. Set	❍	❍	
Organization		❍	❍
Technology			❍

Improving CPI using caches

- An example



What is the improvement (speedup) in memory access time? :

- Caching works because of the principle of locality:
  - Locality found in memory access instructions
    - Temporal locality: if an item is referenced, it will tend to be referenced again soon
    - Spatial locality: if an item is referenced, items whose addresses are close by tend to be referenced soon
  - 90/10 locality rule
    - A program executes about 90% of its instructions in 10% of its code
  - We will look at how this principle is exploited in various microarchitecture techniques

Computer Architecture

计 算 机 体 系 结 构

第2讲：ISA and ILP（1）

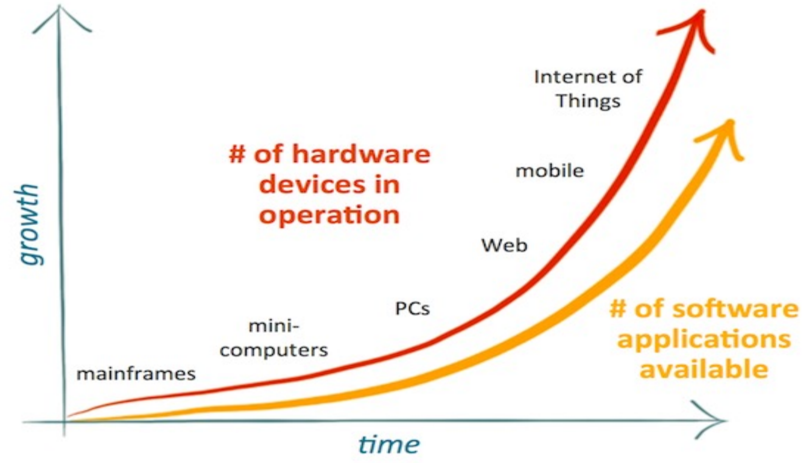
张献伟

xianweiz.github.io

DCS3013, 9/14/2022

The History

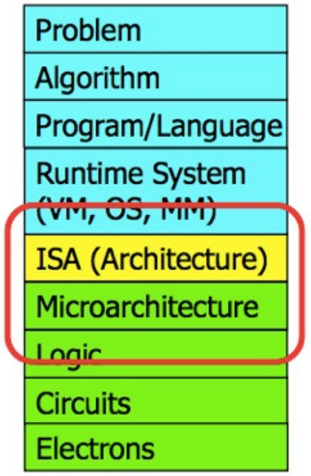
- For more than 50 years, we have enjoyed exponentially increasing compute power[算力急剧增长]
- The growth is based on a fundamental contract between HW and SW[得益于软硬件之间的协议]
  - HW may change radically “under the hood”
    - Old SW can still run on new HW (even faster)
  - HW looks the same to SW, always speaking the same language
    - The **ISA**, allows the decoupling of SW development from HW dev



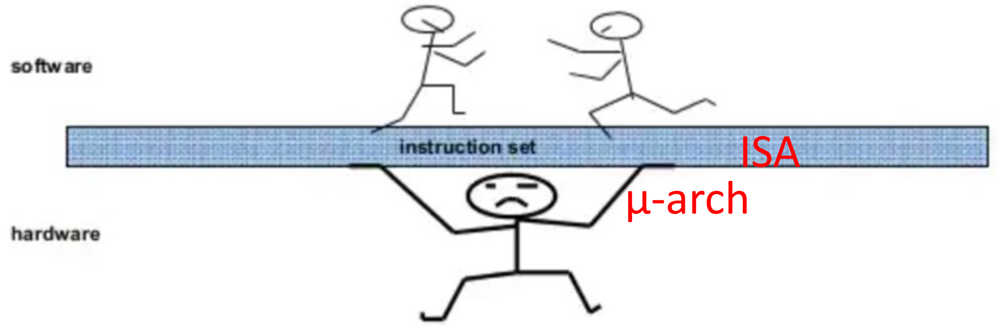
What is ISA?

- Instruction Set == A set of instructions
- The HW/SW contract[软硬件协议]
  - Compiler correctly translates source code to the ISA[编译器]
  - Assembler translates to relocatable binary[汇编器]
  - Linker solidifies relocatables into object code[连接器]
  - HW promises to do what the object code says[硬件执行]
- Not in the “contract”: non-functional aspects[非协议]
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less

ISA + μ-arch = Arch



- "Architecture" = ISA + microarchitecture
- ISA[指令集架构]
  - Agreed upon interface between sw and hw
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture (μ-arch)[微架构]
  - Specific implementation of an ISA
    - Implementation of the ISA under specific design constraints and goals
  - Not visible to the software

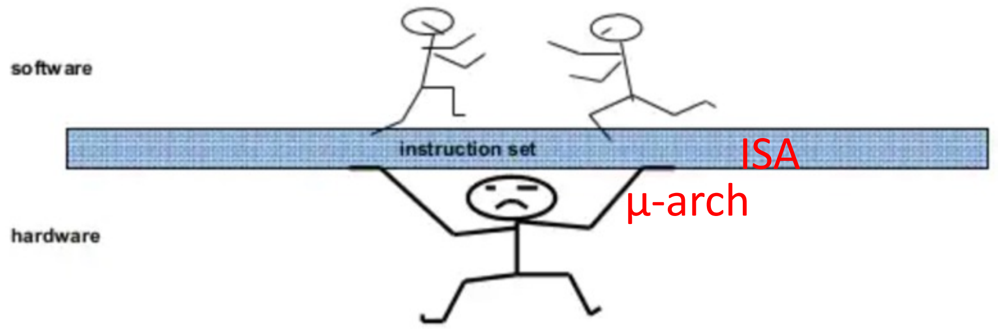


https://image.slideserve.com/466455/instruction-set-design-l.jpg

ISA vs. μ-arch (cont.)

- Implementation (μ-arch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, …

- μ-arch usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many u-archs

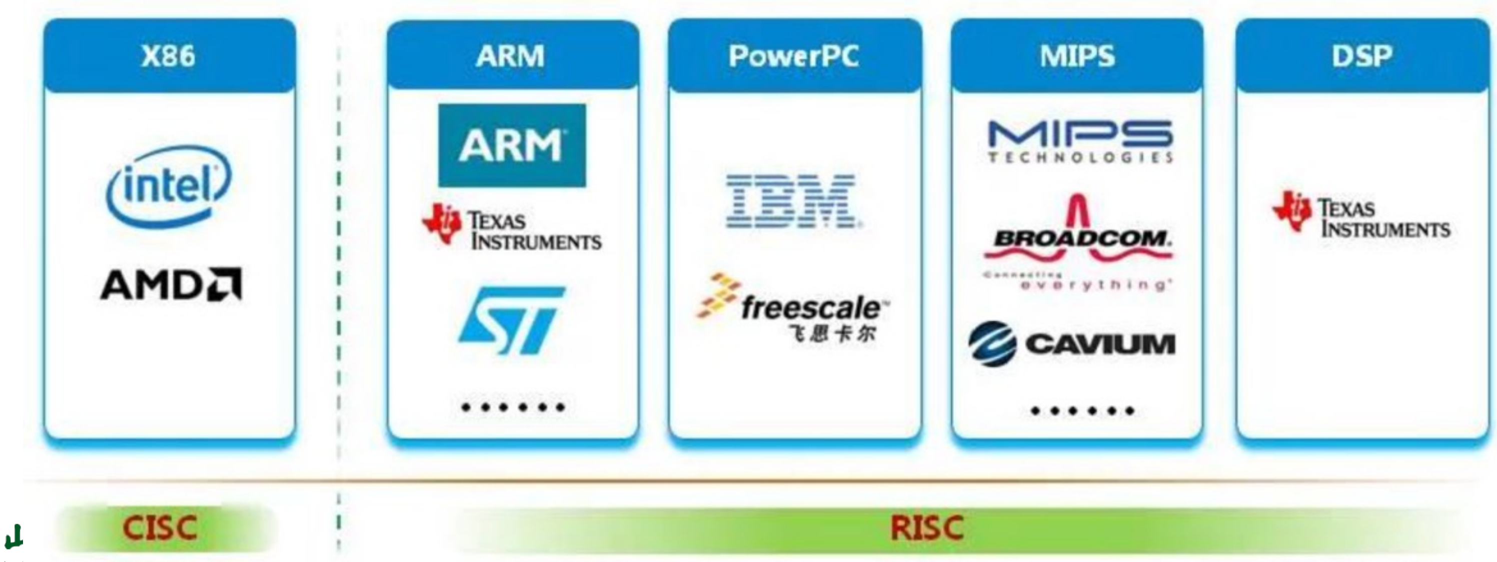


What Makes a Good ISA?

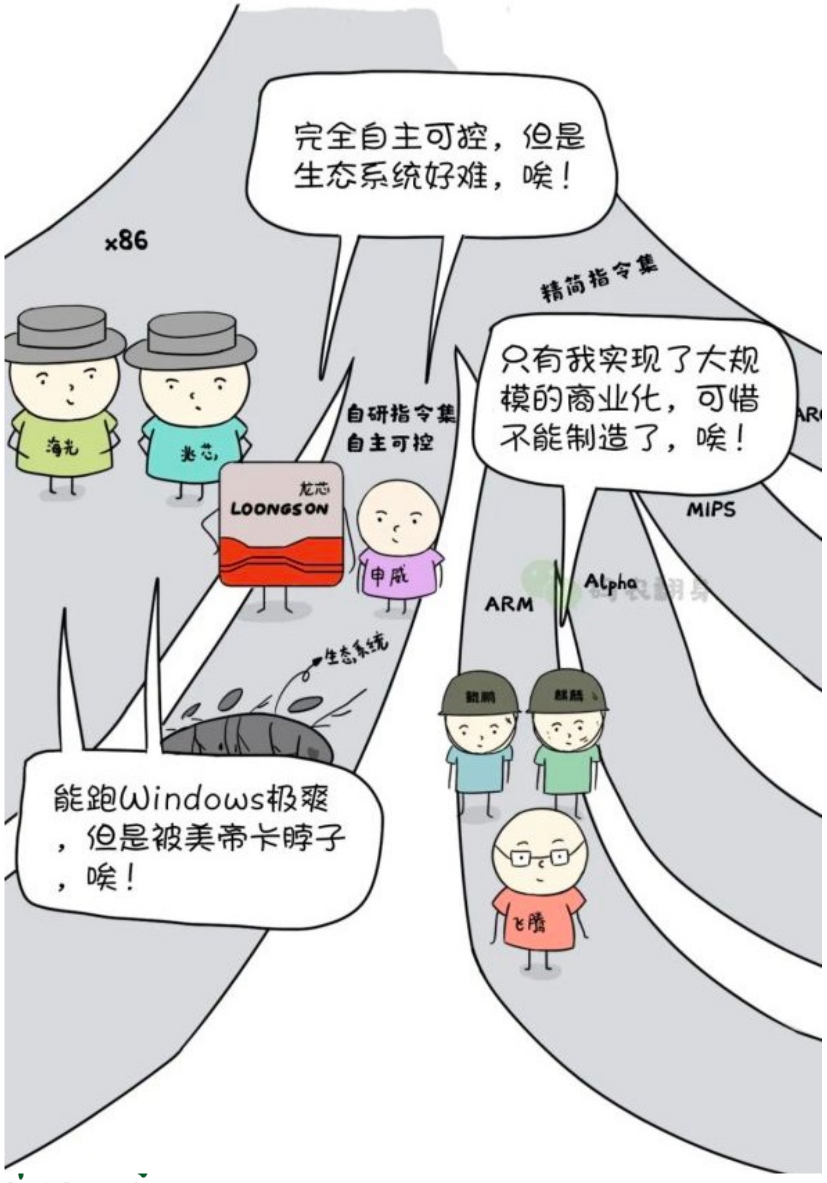
- Programmability[可编程性]
  - Easy to express programs efficiently?
- Implementability[可实现性]
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- Compatibility[兼容性]
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2…

Existing ISAs

- RISC: reduced-instruction set computer[精简指令集]
  - Coined by Patterson in early 80's
  - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- CISC: complex-instruction set computer[复杂指令集]
  - Term didn't exist before "RISC"
  - Examples: x86, VAX, Motorola 68000, etc.



国产架构


完全自主可控，但是生态系统好难，唉！
x86
精简指令集
只有我实现了大规模的商业化，可惜不能制造了，唉！
海光
兆芯
龙芯 LOONGSON
申威
自研指令集 自主可控
MIPS
ARM
Alpha
生态系统
能跑Windows极爽，但是被美帝卡脖子，唉！
飞腾

- x86
  - 曙光/海光
- ARM
  - 华为、飞腾
- 自主
  - 龙芯、申威

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)