11. Performance Engineering

Game Engine Basics

Prof. H. Kang

Introduction

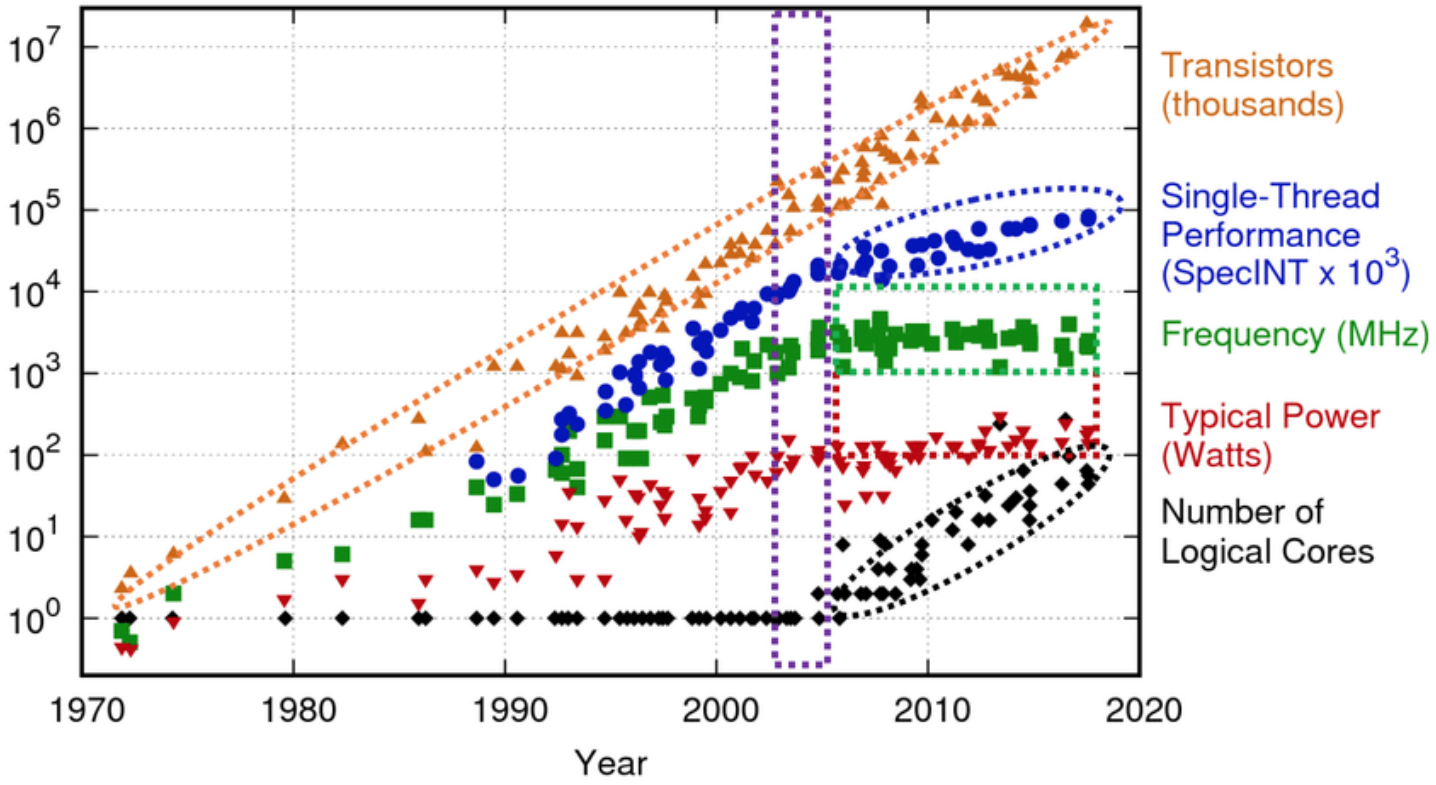
Dennard Scaling

- Computer engineers, aiming to maximize performance, essentially just choose the maximum possible clock rate so that the overall power consumption stays the same.
- If transistors become smaller, they have less capacitance, meaning less required voltage to flip them, which in turn allows increasing the clock rate.

Introduction

Dennard Scaling

- Around 2007, this strategy stopped working because of leakage effects:
  - The circuit features became so small that their magnetic fields started to make the electrons in the neighboring circuitry move in directions they are not supposed to, causing unnecessary heating and occasional bit flipping.



Introduction

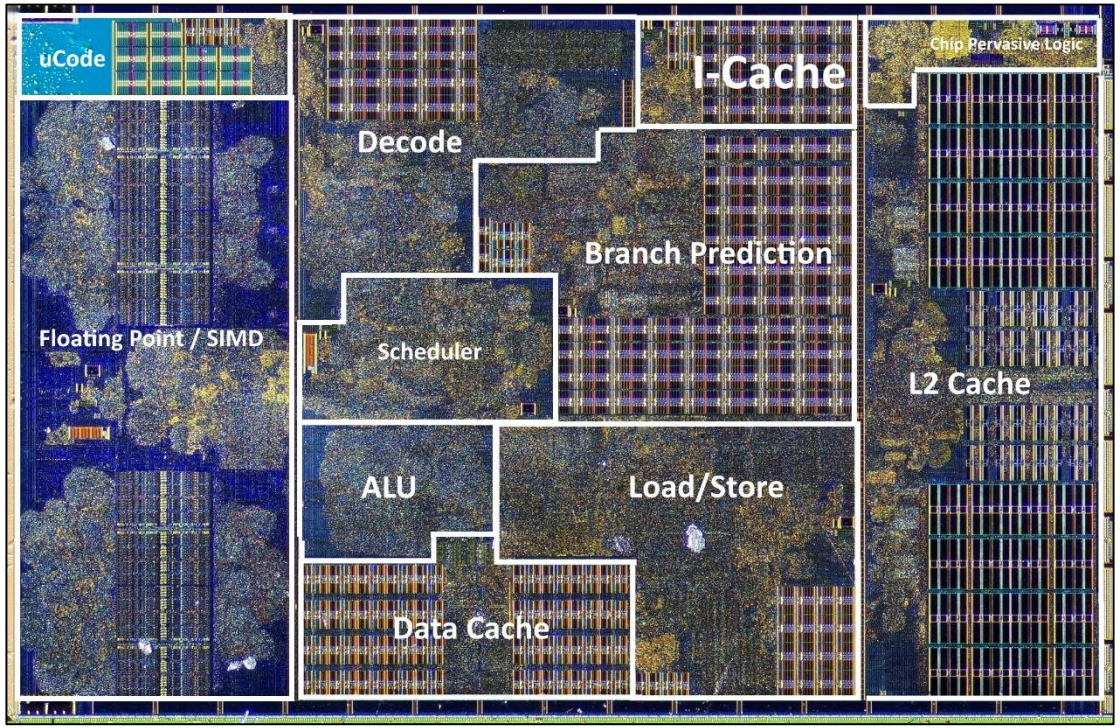
Dennard Scaling

- The only way to mitigate this is to increase the voltage:
  - Then, to balance off power consumption, clock frequency should be reduced, which in turn makes the whole process progressively less profitable as transistor density increases.
  - At some point, clock rates could no longer be increased by scaling, and the miniaturization trend started to slow down.

Introduction

Modern Computing

- Clock rates plateaued, but the transistor count is still increasing, allowing for the creation of new, parallel hardware.
  - Instead of chasing faster cycles, CPU designs started to focus on getting more useful things done in a single cycle.
  - Instead of getting smaller, transistors have been changing shape.
- This resulted in increasingly complex architectures capable of doing dozens, hundreds, or even thousands of different things every cycle.



Instruction-level Parallelism

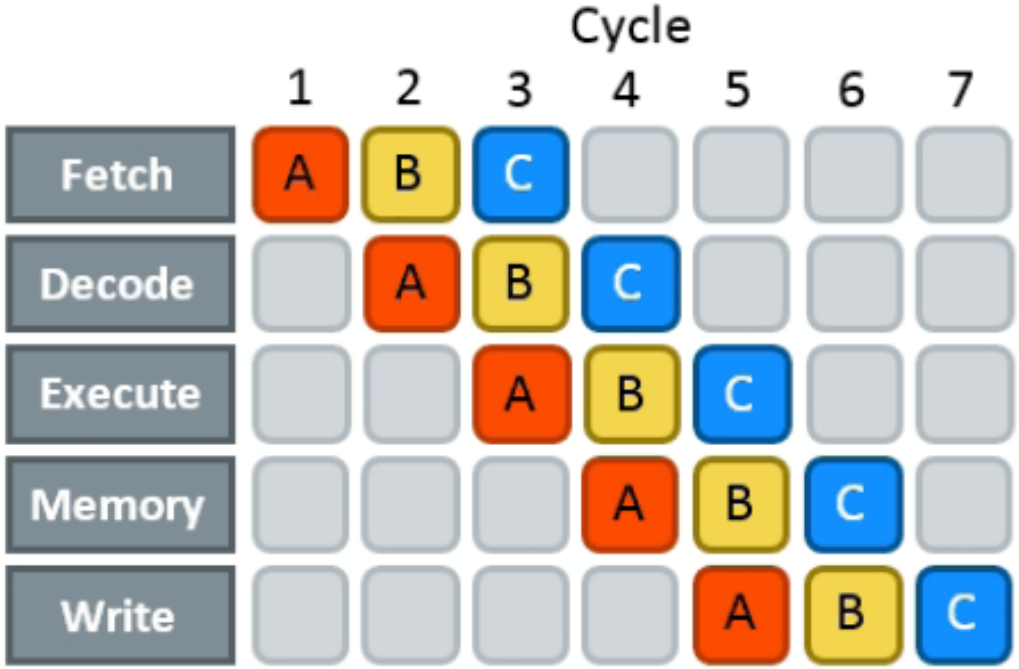
Instruction Pipelining

- To execute any instruction, processors need to do a lot of preparatory work first, which includes:
  - **Fetching** a chunk of machine code from memory,
  - **Decoding** it and splitting into instructions,
  - **Executing** these instructions, which may involve doing some memory operations,
  - **Writing** the results back into registers.

Instruction-level Parallelism

Instruction Pipelining

- This whole sequence of operations is long.
- It takes up to 15-20 CPU cycles even for something simple like add-ing two register-stored values together.
- To hide this latency, modern CPUs use pipelining: after an instruction passes through the first stage, they start processing the next one right away, without waiting for the previous one to fully complete.



Instruction-level Parallelism

Pipeline Hazards

- Pipelining lets you hide the latencies of instructions by running them concurrently, but also creates some potential obstacles of its own — characteristically called pipeline hazards, that is, situations when the next instruction cannot execute on the following clock cycle.
- There are multiple ways this may happen:
  - A structural hazard happens when two or more instructions need the same part of CPU (e.g., an execution unit).
  - A data hazard happens when you have to wait for an operand to be computed from some previous step.
  - A control hazard happens when a CPU can’t tell which instructions it needs to execute next.

Instruction-level Parallelism

Pipeline Hazards

- Different hazards have different penalties:
  - In structural hazards, you have to wait (usually one more cycle) until the execution unit is ready. They are fundamental bottlenecks on performance and can't be avoided — you have to engineer around them.
  - In data hazards, you have to wait for the required data to be computed (the latency of the critical path). Data hazards are solved by restructuring computations so that the critical path is shorter.
  - In control hazards, you generally have to flush the entire pipeline and start over, wasting a whole 15-20 cycles. They are solved by either removing branches completely, or making them predictable so that the CPU can effectively speculate on what is going to be executed next.

Instruction-level Parallelism

The Cost of Branching

- When a CPU encounters a conditional jump or any other type of branching, it doesn't just sit idle until its condition is computed — instead, it starts speculatively executing the branch that seems more likely to be taken immediately.
- During execution, the CPU computes statistics about branches taken on each instruction, and after some time, they start to predict them by recognizing common patterns.
- For this reason, the true "cost" of a branch largely depends on how well it can be predicted by the CPU. If it is a pure 50/50 coin toss, you have to suffer a control hazard and discard the entire pipeline, taking another 15-20 cycles to build up again. And if the branch is always or never taken, you pay almost nothing except checking the condition.

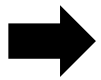
```
int main() {
    int a = 5;
    int b = 20;
    if (a < b) {
        a = 10;
    }
    return a;
}
```

Instruction-level Parallelism

Branchless Programming

- Branches that can't be effectively predicted by the CPU are expensive as they may cause a long pipeline stall to fetch new instructions after a branch mispredict.
- To optimize this, conditionals should be removed.
  - However, they are the inevitable part of a program.
  - Therefore, not every scenario can be optimized.
  - In some cases replacing branches will degrade the performance as it might introduce more additional instructions.

```
if(a > b) {
 return a;
} else {
 return b;
}
```



```
return (a > b) * a + (a <= b) * b;
```

```
int fast_max(int a, int b) {
  int diff = a - b;
  int dsgn = diff >> 31;
  return a - (diff & dsgn);
}
```

Reference

[1] https://en.algorithmica.org/hpc/pipelining/branchless/
[2] https://dev.to/jobinrjohnson/branchless-programming-does-it-really-matter-20j4