CPU Efficient Code

Holger Pirk

Slides as of 18/01/21 13:31:21

Bottleneck

Definition: Bottleneck

The resource with the highest utilization

Challenge:

Can we build a system without a bottleneck – one where all resources are equally utilized?

A first attempt at balance

> *A balanced computer system needs 1 MB of main memory capacity and 1Mbit per second of I/O bandwidth per MIPS of CPU performance. – Gene Amdahl (paraphrased by Hennessy & Patterson)*

- Do we buy into that?

Let's calculate…

- My laptop's CPU has

```
IPS = 2.9 (* GHz *) * 4 (* Cores *) * 4 (* Execution slots/cycle *)

46.4
```

```
Bandwidth = 37.5 (* GB/s *) * 8 (* bits/byte *)

300
```

```
Memory = 16 (* GB *)
```

Verdict this thing is **way** underpowered in terms of CPU

- Who would design a system like that?

Balance is not a constant, . . .

- Datatypes are larger today than they were in Amdahl's time
- CPUs have all kinds of Co-processors (on-die GPUs, DMA engines, . . .)
- Cache lines are larger, causing wasted memory bandwidth,
- etc.
- **However**, . . .

. . . balance is a function

- Hardware optimizations have varying impact on code
- There is no such thing as a balanced system (not even

balanced applications), only balanced sections of code

- We distinguish balanced, compute-bound, latency-bound and

bandwidth-bound code

Amdahl got one thing right…

- The fundamental tradeoff is between CPU and Bandwidth efficiency

You can influence bottlenecks

- High-level techniques:
  - Choice of algorithm
  - Memoization
  - Compression
  - ...
- Low-level techniques
  - Those are our focus

Let's focus on compute-bound code for today

When CPU-efficiency matters

- For CPU-bound applications, i.e., those that
  - are poorly implemented
  - operate on small (cache-resident) datasets
  - are math-heavy (especially floating point math)
  - apply memory-oriented optimizations (let's discuss those on Friday)

CPU-efficiency: target metric

- Well, wallclock time, obviously
- Cycles per Instruction (CPI) is not that useful as an optimization metric since the instructions are a parameter (
- Stall cycles are a good indicator (though not perfect either)
- Stall cycles are caused by hazards:
    - Control-hazards - stalls due to data-dependent changes in the control flow
    - Structural hazards - stalls due to the lack of execution resources (registers, execution ports, . . .)
    - Data-hazards - stalls due to operands (i.e., data) not being available on time

CPU-efficiency: proxy metrics

Work-efficiency Don’t waste cycles on unnecessary work

Simplicity Fewer instructions occupy less resources (caches, registers, execution slots)

Independent Instructions Allows the CPU to fill pipeline bubbles

Parallelism Parallelism matters even if you don’t write parallel algorithms

Predictability CPUs speculate for performance, make your code behave accordingly

Adaptivity Account for parameters you don’t know the value of

Specialization Use the features of your CPU

Separation of Concerns Make the hardware do the heavy lifting

But: do they all matter the same?

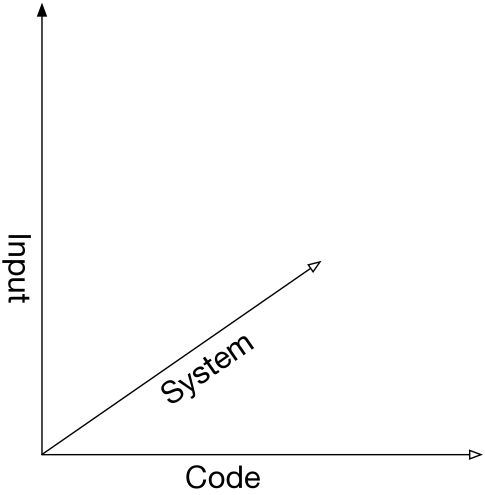
Discussion:

Which is more important: Parallelism or Predictability?

The answer is, of course,. . .

. . . it depends

Problem parameter dimensions



The System

- Fundamental design decisions:
    - Is the system executing instructions out-of-order
    - Is the system executing instructions speculatively
    - Is the system executing work in parallel
        - Dynamically bundled packages: superscalar execution
        - Statically bundled packages (SIMD or VLIW)

Pipelined Execution exploits independence of stages of different instructions

	T1	T2	T3	T4	T5	T6	T7	T8	T9
1	Fetch	Decode	Execute	Load	Write				
2		Fetch	Decode	Execute	Load	Write			
3			Fetch	Decode	Execute	Load	Write		
4				Fetch	Decode	Execute	Load	Write	
5					Fetch	Decode	Execute	Load	Write

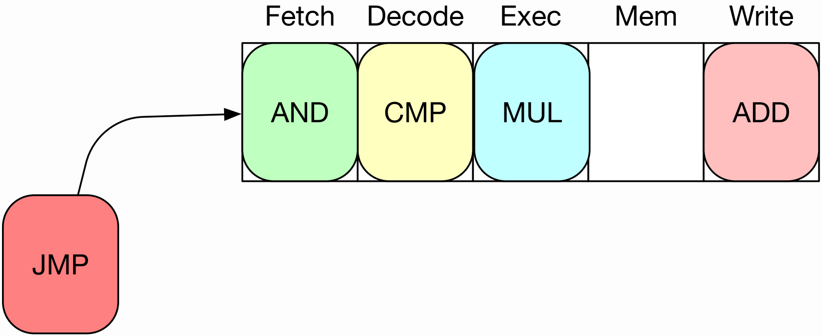
Suffers from all three hazards: control, data and structural

Out-of-order execution exploits independence of the instructions

- Kicks in after decoding decode-phase
- Instructions that have neither unsatisfied structural- nor data-dependencies move on to execution stage
- Examples:
  - In the statement `a*b+d*e+b*b`, src_c++[:exports code]{b*b} is scheduled before `d*e` when `b` is L1 cache-resident
- Evaluate an integer addition even if there are floating point additions waiting for execution ports
- Addresses data and structural hazards, leaves control hazards

Pipelined Execution exploits independence of stages of different instructions

An ALU Stall



Speculative execution

- Keeps the pipeline filled if no instructions are eligible
- Addresses control hazards

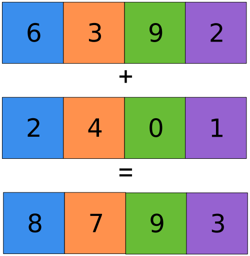
Superscalar Execution

	T1	T2	T3	T4	T5	T6	T7	T8	T9
1	Fetch	Decode	Execute	Load	Write				
2	Fetch	Decode	Execute	Load	Write				
3		Fetch	Decode	Execute	Load	Write			
4		Fetch	Decode	Execute	Load	Write			
5			Fetch	Decode	Execute	Load	Write		
6			Fetch	Decode	Execute	Load	Write		
7				Fetch	Decode	Execute	Load	Write	
8				Fetch	Decode	Execute	Load	Write	
9					Fetch	Decode	Execute	Load	Write
10					Fetch	Decode	Execute	Load	Write

Pipelined Execution Different instructions can be in different stages

Superscalar Execution Different instructions can be in the same stage

SIMD



Single Instruction Multiple Data

- The CPU performs the same operation on multiple data items
- Introduced for multimedia but useful for general computing
- Applications range from the trivial to the mind-blowing
- Every CPU-generation has new instructions: MMX, SSE, SSE2, AVX,

AVX2, AVX512

- They use specialized vector-registers

A sidenote on **VLIW**

- SIMD means
  - single instruction
  - the same operation on multiple data items
- **V**ery **L**ong **I**nstruction **W**ords
  - single instruction
  - different operations on multiple or the same data item
  - Basically compiler-controlled superscalar execution

Rule: Write **runtime** efficient code

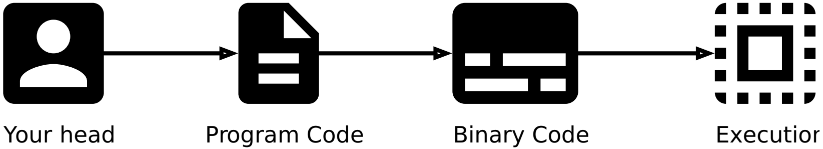
- Do not run code repeatedly – duh!
- Evaluate code as early as possible

What do I mean by that?

Partial evaluation (make the compiler work for you)

- Idea: Treat programs evaluation as a multi-phased process
    - In every phase, you know more about the result
    - i.e., you can evaluate more of it/create a more specialized program
    - the fix point is obviously the actual result

The code execution process



Examples of partial evaluation

- Function inlining
- JiT compilation
- Symbolic Programming
- Constant evaluation
    - Careful: when are things truly constants?

Constants

```
int result(int input) {
        return input*3*5;
};

int result2(int input) {
        int three = 3-1;
        int five = 4+1;
        return input*three*five;
};

int three = 3;
int five = 5;
int result3(int input) {
        return input*three*five;
};
```

- see `https://godbolt.org/z/X3nQb2`
- What constitutes a constant depends on your language's semantics

Lifting expensive operations

Lifting Moving work from a section that is executed often into one that is executed seldomly

- Standard example moving invariants out of a loop:
  - `for (size_t i = 0; i < N; i++) output = 7*8;`
  - → `int tmp = 7*8; for (size_t i = 0; i < N; i++) output = tmp;`
    - Can be generalized to *loop specialization*

Loop specialization

Example

```c
#include <stdlib.h>
void scaleVector(int* input, size_t inputSize, int scale) {
        for(size_t i = 0; i < inputSize; i++)
                input[i] *= scale;
};

void scaleVectorSmart(int* input, size_t inputSize) {
        if(scale != 1) {
                if(scale == 2)
                        for(size_t i = 0; i < inputSize; i++)
                                input[i] <<= 1;
                else
                        for(size_t i = 0; i < inputSize; i++)
                                input[i] *= scale;
        }
};
```

The problem with loop specialization

- Leads to code duplication
- You can do it by hand but. . .
- the compiler could easily do these for you but

How many special cases should it generate?

- Compilers need help
- Introducing: metaprogramming. . .

Metaprogramming – . . . what you can do for your compiler

The idea

- Generating all these special cases at compile-time
- Allows the compiler to apply optimizations for the special cases

Implementations

- Macros in LISP (and it's dialects), Haskell, Scala
- Templates in C++, Macros in C (they are called the same but aren't)
- Generics in Python, Java, C#

Metaprogramming – . . . what you can do for your compiler

Turn this code. . .

```
void scaleVector(int* input, size_t inputSize, int scale) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};
int useIt(int* input, size_t size) {
  scaleVector(input, size, 2);
  scaleVector(input, size, 1);
  scaleVector(input, size, 0);
}
```

. . . into this

```
template <int scale> void scaleVectorPE(int* input, size_t inputSize) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};
int useIt(int* input, size_t size) {
  scaleVectorPE<2>(input, size);
  scaleVectorPE<1>(input, size);
  scaleVectorPE<0>(input, size);
}
```

Metaprogramming – . . . what you can do for your compiler

Same example

```
#include <functional>
#include <map>
#include <stdlib.h>
template <int scale> void scaleVectorPE(int* input, size_t inputSize) {
  for(size_t i = 0; i < inputSize; i++)
    input[i] *= scale;
};

void scaleVector(int* input, size_t inputSize, int scale) {
  std::map<int, std::function<void(int*, size_t)>> const specialCases{{0, scaleVectorPE<0>},
  ↪  //
                                                                      {1, scaleVectorPE<1>},
                                                                      ↪  //
                                                                      {2, scaleVectorPE<2>},
                                                                      ↪  //
                                                                      {4,
                                                                      ↪  scaleVectorPE<4>}};
  if(specialCases.count(scale))
    specialCases.at(scale)(input, inputSize);
  else
    for(size_t i = 0; i < inputSize; i++)
      input[i] *= scale;
};
```

Control-dependencies **can** cause control hazards

```
for(size_t i = 0; i < inputSize; i++)
        if(input[i] < high)
                output[outI++] = input[i];
```

Branch-free code **can** eliminate control hazards

```
for(size_t i = 0; i < inputSize; i++) {
  output[outI] = input[i];
  outI += (input[i] < high);
}
```

- This is called predication or if-conversion depending on who you ask

Control-dependencies **might not** cause control hazards

- if the input is predictable

```
--------------------------------------------------------------
Benchmark                       Time           CPU Iterations
--------------------------------------------------------------
selection/50             49951160 ns   49951326 ns         12
selection/100            52085927 ns   52085751 ns         10
selection/150            54732263 ns   54731816 ns         13
selection/200            57072435 ns   57072634 ns         12
selection/250            59132204 ns   59132429 ns         11
selection/300            62251051 ns   62250581 ns         11
selection/350            66982496 ns   66982226 ns         10
selection/400            67639907 ns   67640118 ns          9
selection/450            70828641 ns   70828503 ns          9
selection/500            73505346 ns   73505682 ns          9
selection/550            73040486 ns   73040182 ns          9
```

- Bottom line: Predictability is input-dependent

Guideline

- Branch-predictors are the way they are for a reason
- Lots of code is already very predictable
- Predication/if-conversion turns control dependencies into data dependencies
    - Data dependencies cause data hazards, predication amplifies these
    - only apply after measuring

SIMD vectorization

- Compilers try to auto-vectorize
  - They succeed for simple cases like this
    - `for (size_t i = 0; i < 1024; i++) out[i] = in1[i]*in2[i]`
  - Write code with vectorization in mind, godbolt your code to be sure
- If they fail: explicitly vectorize using intrinsics
- Intrinsics are c-functions that map directly to hardware instructions:
  - `_mm256_<intrin_op>_<suffix>(<data type> <parameter1>, <data type> <parameter2>, <data type> <parameter3>)`
  - Check out this page:
    - https://software.intel.com/sites/landingpage/IntrinsicsGuide

SIMD: Example

```
#include <immintrin.h>
#include <benchmark/benchmark.h>
#include <random>
#include <iostream>
using namespace std;

union v8f {
        float floats[8];
        __m256 simdVector;
};
```

SIMD: Example

```cpp
        auto bounds1 = state.range(0);
        auto bounds2 = state.range(1);
        auto input1 = new int[bounds1];
        auto input2 = new float[bounds2];

        uniform_int_distribution<mt19937::result_type> randomInRange(0, bounds2);
        for(size_t i = 0; i < bounds1; i++)
                input1[i] = randomInRange(rng);
        for(size_t i = 0; i < bounds2; i++)
                input2[i] = randomInRange(rng);

        for(auto _ : state) {
                float sum = 0;
                for(size_t i = 0; i < bounds1; i++) {
                        sum += input2[input1[i]];
                }
// ...
```

SIMD: Example

```
float sum = 0;
for(size_t i = 0; i < bounds1 / 8; i++) {
        v8f values{.simdVector = _mm256_i32gather_ps(input2,
        ↪  ((__m256i*)input1)[i], sizeof(int))};
        for(size_t i = 0; i < 8; i++)
                sum += values.floats[i];
}
```

SIMD: Example

```
v8f sums{};
for(size_t i = 0; i < bounds1 / 8; i++) {
        v8f values{.simdVector = _mm256_i32gather_ps(input2,
        ↪  ((__m256i*)input1)[i], sizeof(int))};
        sums.simdVector = _mm256_add_ps(values.simdVector, sums.simdVector);
}
float sum = 0;
for(size_t i = 0; i < 8; i++)
        sum += sums.floats[i];
```

Bottom line

- Use intrinsics for hard to auto-vectorize code
- Try to keep data in vector registers
- Godbolt your code