Writing Efficient Code in C(++)

Petr Ročkai

November 11, 2019

Organisation

- theory: 20-30 minutes every week
- coding: all the remaining time
- passing the subject: collect 15 points
- most points come from assignments
- showing up 5 times gets you 1 point

Assignments

- one assignment every 2 weeks, 6 in total
- missing the deadline is the same as failing it
- one assignment = 2 points

Bonuses (per assignment)

- add 1 point if you pass within 14 days
- else add 0.5 points if you pass by 20.12.

Assignments (cont'd)

- details about submission next week
- you can use C or C++
- the code must be valid C11 or C++17
- lite tests run every midnight
- full tests and reviews are done at deadlines

Competitions

- we will hold 3 competitions in the seminar
- do your best in 40 minutes on a small problem
- the winner gets 2 points, second place gets 1 point
- all other working programs get 0.5 points
- we'll dissect the winning program together

Exercises in Seminar

- one of you will be programming live
- i.e. what you do will be shown on the beamer
- you have to do this once to pass the subject
  - it is okay to do it more than once though
  - you will get 1 point for each instance

Semester Plan (part 1)

		date
1.	computational complexity	16.9.
2.	microbenchmarking & stats	23.9.
3.	the memory hierarchy	30.9.
	cancelled	7.10.
4.	using `callgrind`	14.10.
5.	tuning for the compiler	21.10.
	state holiday	28.10.

Semester Plan (part 2)

		date
6.	competition 1	4.11.
7.	understanding the CPU	11.11.
8.	exploiting parallelism	18.11.
9.	competition 2	25.11.
10.	using `perf`	2.12.
11.	Q&A, homework recap	9.12.
12.	competition 3	16.12.

Assignment Schedule

		given	due
1.	benchmarking tool	23.9.	7.10.
2.	matrix multiplication	7.10.	21.10.
3.	sets of integers	21.10.	4.11.
4.	substring search	4.11.	18.11.
5.	parallel computation	18.11.	2.12.
6.	a hash table	2.12.	16.12.

Efficient Code

- computational complexity
- the memory hierarchy
- tuning for the compiler & optimiser
- understanding the CPU
- exploiting parallelism

Understanding Performance

- writing and evaluating benchmarks
- profiling with `callgrind`
- profiling with `perf`
- the law of diminishing returns

- premature optimisation is the root of all evil
- (but when is the right time?)

Tools

- on a POSIX operating system (preferably not in a VM)
- `perf` (Linux-only, sorry)
- `callgrind` (part of the `valgrind` suite)
- `kcachegrind` (for visualisation of `callgrind` logs)
- maybe `gnuplot` for plotting performance data

Compilers

- please stick to C++17 and C11 (or C99)
- the reference compiler will be clang 8.0.0
- you can use other compilers locally
- but your code has to build with clang 8

Part 1: Computational Complexity

Complexity and Efficiency

- this class is not about asymptotic behaviour
- you need to understand complexity to write good code
- performance and security implications

- what is your expected input size?
- complexity vs constants vs memory use

Quiz

- what's the worst-case complexity of:
  - a bubble sort? (standard) quick sort?
  - inserting an element into a RB tree?
  - inserting an element into a hash table?
  - inserting an element into a sorted array?
  - appending an element to a dynamic array?
- what are the amortised complexities?
- how about expected (average)?

Hash Tables

- often the most efficient data structure available
- poor theoretical worst-case complexity
  - what if the hash function is really bad?
- needs a fast hash function for efficiency
  - rules out secure (cryptographic) hashes

Worst-Case Complexity Matters

- CVE-2011-4815, 4838, 4885, 2012-0880, ...
- apps can become unusable with too many items
- use a better algorithm if you can (or must)
- but: simplicity of code is worth a lot, too
- also take memory complexity and constants into account

Constants Matter

- n ops if each takes 1 second
- $n \log n$ ops if each takes .1 second
- n^2 ops if each takes .01 second

Picking the Right Approach

- where are the crossover points?
- what is my typical input size?
- is it worth picking an approach dynamically?
- what happens in pathological cases?

Exercises

- log into `aisa`
- run `pb173eff update`
- then `cd ~/pb173eff/01`
- and `cat intro.txt`

Part 2: Microbenchmarking & Statistics

Motivation

- there's a gap between high-level code and execution
- the gap has widened over time
  - higher-level languages & more abstraction
  - more powerful optimisation procedures
  - more complex machinery inside the CPU
  - complicated cache effects
- it is very hard to predict actual performance

Challenges

- performance is very deterministic in theory
- this is not the case in practice
  - time-sharing operating systems
  - cache content and/or swapping
  - power management, CPU frequency scaling
  - program nondeterminism; virtual machines
- both micro (unit) and system benchmarks are affected

Unit vs System Benchmarking

- a benchmark only gives you one number
- it is hard to find causes of poor performance
- unit benchmarks are like unit tests
  - easier to tie causes to effects
  - faster to run (minutes or hours vs hours or days)
  - easier to make parametric

Isolation vs Statistics

- there are many sources of measurement errors
- some are systematic, others are random (noise)
- noise is best fought with statistics
- but statistics can't fix systematic errors
- benchmark data is not normally distributed

Repeated Measurements

- you will need to do repeat measurements
- more repeats give you better precision
  - the noise will average out
  - execution time vs precision tradeoff
- the repeat runs form your input sample
  - this is what you feed into bootstrap

Bootstrap

- usual statistical tools are distribution-dependent
- benchmark data is distributed rather oddly
- idea: take many random re-samplings of the data
- take 5th and 95th percentile as a confidence interval
- this is a very robust (if stochastic) approach

Implementing Bootstrap

- inputs: a sample, an estimator and iteration count
- outputs: a new sample
- in each iteration, create a random resample
  - add a random item from the original sample
  - repeats are allowed (this is important)
  - size of the resample = size of the original

Estimators

- most useful estimators are the mean (average)
- and various percentiles (e.g. median)
- you can also estimate standard deviation
  - but keep in mind the original data is not normal

Output Distribution

- the output of bootstrap is another distribution
- you can expect this one to be normal
- it is the distribution of the estimator result
- you can compute the mean and σ of the bootstrap

Part 3: The Memory Hierarchy

- CPU registers: very few, very fast (no latency)
- L1 cache: small (100s of KiB), plenty fast (~4 cycles)
- L2 cache: still small, medium fast (~12 cycles)
- L3 cache: ~2-32 MiB, slow-ish (~36 cycles)
- L4 cache: (only some CPUs) ~100 MiB (~90 cycles)
- DRAM: many gigabytes, pretty slow (~200 cycles)

- NVMe: ~10k cycles
- SSD: ~20k cycles
- spinning rust: ~30M cycles
- RTT to US: ~450M cycles

Paging vs Caches

- page tables live in slow RAM
- address translations are very frequent
- and extremely timing-sensitive
- TLB $→$ small, very fast address translation cache

- process switch → TLB flush
- but: Tagged TLB, software-managed TLB
- typical size: 12 - 4k entries
- miss penalties up to 100 cycles

Additional Effects

- some caches are shared, some are core-private
- out of order execution to avoid waits
- automatic or manual (compiler-assisted) prefetch
- speculative memory access
- ties in with branch prediction

Some Tips

- use compact data structures (`vector` beats `list`)
- think about locality of reference
- think about the size of your working set
- code size, not just speed, also matters

See Also

- `cpumemory.pdf` in study materials
  - somewhat advanced and somewhat long
  - also very useful (the title is not wrong)
  - don't forget to add 10 years
  - oprofile is now `perf`
- http://www.7-cpu.com CPU latency data

Part 4: Profiling I, callgrind

Why profiling?

- it's not always obvious what is the bottleneck
- benchmarks don't work so well with complex systems
- performance is not quite composable
- the equivalent of `printf` debugging isn't too nice

Workflow

1. use a profiler to identify expensive code
   - the more time program spent doing X,
   - the more sense it makes to optimise X
2. improve the affected section of code
   - re-run the profiler, compare the two profiles
   - if satisfied with the improvement, goto 1
   - else goto 2

What to Optimise

- imagine the program spends 50 % time doing X
  - optimise X to run in half the time
  - the overall runtime is reduced by 25 %
  - good return on investment
- law of diminishing returns
  - now only 33 % of time is spent on X
  - cutting X in half again only gives 17 % of total
  - and so on, until it makes no sense to optimise X

Flat vs Structured Profiles

- flat profiles are easier to obtain
- but also harder to use
  - just a list of functions and cost
  - the context & structure is missing
- call stack data is a lot harder to obtain
  - endows the profile with very rich structure
  - reflects the actual control flow

`cachegrind`

- part of the `valgrind` tool suite
- dynamic translation and instrumentation
- based on simulating CPU timings
  - instruction fetch and decode
  - somewhat abstract cost model
- can optionally simulate caches
- originally only flat profiles

`callgrind`

- records entire call stacks
- can reconstruct call graphs
- very useful for analysis of complex programs

`kcachegrind`

- graphical browser for callgrind data
- demo

Part 5: Tuning for the Compiler

Goals

- write high-level code
- with good performance

What We Need to Know

- which costs are easily eliminated by the compiler?
- how to best use the optimiser (with minimal cost)?

How Compilers Work

- read and process the source text
- generate low-level intermediate representation
- run IR-level optimisation passes
- generate native code for a given target

Intermediate Representation

- for C++ compilers typically a (partial) SSA
- reflects CPU design / instruction sets
- symbolic addresses (like assembly)
- explicit control and data flow

IR-Level Optimiser

- common sub-expression elimination
- loop-invariant code motion
- loop strength reduction
- loop unswitching
- sparse conditional constant propagation
- (regular) constant propagation
- dead code elimination

Common Sub-expression Elimination

- identify redundant (& side-effect free) computation
- compute the result only once & re-use the value
- not as powerful as equational reasoning

Loop-Invariant Code Motion

- identify code that is independent of the loop variable
- and also free of side effects
- hoist the code out of the loop
- basically a loop-enabled variant of CSE

The Cost of Calls

- prevents CSE (due to possible side effects)
- prevents all kinds of constant propagation

Inlining

- removes the cost of calls
- improves all intra-procedural analyses
- inflates code size
- only possible if the IR-level definition is available

See also: link-time optimisation

The Cost of Abstraction: Encapsulation

- API or ABI level?
- API: cost quickly eliminated by the inliner
- ABI: not even LTO can fix this
- ABI-compatible setter is a call instead of a single store

The Cost of Abstraction: Late Dispatch

- used for `virtual` methods in C++
- indirect calls (through a vtable)
- also applies to C-based approaches (`gobject`)
- prevents (naive) inlining
- compilers (try to) devirtualise calls

Part 6: Understanding the CPU

The Simplest CPU

- in-order, one instruction per cycle
- sources of inefficiency
  - most circuitry is idle most of the time
  - not very good use of silicon
- but it is reasonably simple

Design Motivation

- silicon (die) area is expensive
- switching speed is limited
- heat dissipation is limited
- transistors cannot be arbitrarily shrunk
- “wires” are not free either

The Classic RISC Pipeline

- fetch – get instruction from memory
- decode – figure out what to do
- execute – do the thing
- memory – read/write to memory
- write back – store results in the register file

Instruction Fetch

- pull the instruction from cache, into the CPU
- the address of the instruction is stored in PC
- traditionally does branch “prediction”
  - in simple RISC CPUs always predicts not taken
  - this is typically not a very good prediction
  - loops usually favour taken heavily

Instruction Decode

- not much actual decoding in RISC ISAs
- but it does register reads
- and also branch resolution
  - might need a big comparator circuit
  - depending on ISA (what conditional branches exist)
  - updates the PC

Execute

- this is basically the ALU
  - ALU = arithmetic and logic unit
- computes bitwise and shift/rotate operations
- integer addition and subtraction
- integer multiplication and division (multi-cycle)

Memory

- dedicated memory instructions in RISC
  - load and store
  - pass through execute without effect
- can take a few cycles
- moves values between memory and registers

Write Back

- write data back into registers
- so that later instructions can use the results

Pipeline Problems

- data hazards (result required before written)
- control hazards (branch misprediction)
- different approaches possible
  - pipeline stalls (bubbles)
  - delayed branching
- structural hazards
  - multiple instructions try to use a single block
  - only relevant on more complex architectures

Superscalar Architectures

- more parallelism than a scalar pipeline
- can retire more than one instruction per cycle
- extracted from sequential instruction stream
- dynamically established data dependencies
- some units are replicated (e.g. 2 ALUs)

Out-of-order execution

- tries to fill in pipeline stalls/bubbles
- same principle as super-scalar execution
  - extracts dependencies during execution
  - execute if all data ready
  - even if not next in the program

Speculative Execution

- sometimes it's not yet clear what comes next
- let's decode, compute etc. something anyway
- fills in more bubbles in the pipeline
- but not always with actual useful work
- depends on the performance of branch prediction

Take-Away

- the CPU is very good at utilising circuitry
- it is somewhat hard to write “locally” inefficient code
- you should probably concentrate on non-local effects
  - non-local with respect to instruction stream
  - like locality of reference
  - and organisation of data in memory in general
  - also higher-level algorithm structure