

The Unwritten Rules Of Optimisation

... or, “there’s only so much a compiler/JIT/VM/etc can do for shitty code..”

Adrian Chadd <adrian@creative.net.au>

The Axioms

- The rules!
 - 1. Don't Do It.
 - 2. Don't Do It Yet.
 - 3. Profile Before Optimisation.
- Why!
 - People aren't as smart as they think
 - The resultant code is often ugly

The Axioms: Wrong?

- Limitations of the axioms
 - Can you optimise the code?
 - .. without too much effort?
 - .. without rewriting it from scratch?
 - .. without making it unmanageable?
- .. again, people think they're more clever than they are.

What else?

- Design!
- APIs!
- Algorithm choices!
- Know your platform!
- Know when to trust your compiler..
- .. know when it's a load of crap.
- Profiling!

Design!

- .. started with “structured programming” (Pascal)
- .. continued with object oriented design (Smalltalk, C++, Java)
- .. now includes managed code (C#, etc)

Design: What not to do

- Know what to encapsulate how; don't go overboard
 - .. too many object classes == slow
 - .. people invent “object caching” design pattern layers to cope with this

APIs!

- Design APIs which are flexible enough to cope with a variety of situations
 - eg - Python HTTP library and “POST”
- .. but you can definitely go overboard
 - eg - wide character support in standard libraries

Algorithm Choices!

- Partly about choosing which algorithms to use
- Partly about structuring your code to make it easier to change/replace things without too much effort

Know your platform!

- Computers aren't hypothetical..
- .. but they're taught that way.
- Adrian's suggestion: read 1960s-1990s textbooks on data structures and algorithms
 - .. they specifically cover a variety of algorithm behaviour on various storage types - memory, disk, tape, punch card
 - .. wait. Why the heck would you care?

Know your platform!

- Multi-tiered storage is absolutely applicable in 2010!
- Rough order goes like this:
 - registers, L1 cache, L2 cache, (L3 cache)
 - local memory line (SDRAM/DDR), local memory, remote memory (eg NUMA)
 - local disk (ssd, fast disk, slow disk), tape
 - LAN, campus, WAN, Internets..

Know your platform!

- Algorithm steps - best case vs worst case
 - Eg: BTree vs B+Tree/B*Tree - disk seek times
 - Memory access times aren't linear
 - Random versus sequential access
- Working with latency - pipelining
 - eg NFS/CIFS; TCP performance

What can compilers do?

- Modern desktop CPUs run code out of order, partially in parallel, etc, etc
 - .. except some low power ones- eg Intel Atom
- .. modern languages typically don't let you express these
- Modern compilers attempt to bridge this gap

.. what they can't do?

- They can't optimise poorly written code
- They (somewhat) try to combine lots of abstractions into mostly optimal ones
 - eg C++ templates
- .. but there's a fine line between what code generates good code, and what doesn't
 - .. and that can change with versions!

.. what to do?

- Write simple, clean code
- Don't use language features if you don't have to!
- Worry about less simple, more complicated code later
- C++ example - templates are good, too much template use makes later optimising difficult

Profiling!

- Learn how to profile!
 - MacOS: Shark
 - Linux: OProfile
 - FreeBSD: PMC
 - Solaris (/FreeBSD): DTrace
 - Windows: various commercial options

Profiling: non-CPU?

- Lots of profiling tools focus on CPU only profiling
- .. what about profiling disk? network behaviour?
 - disk: dtrace is very useful
 - network: tcpdump, tcpreplay

Final comments

- The compiler can't completely compensate for shitty code
- Neither can faster CPUs - network, storage?
- Faster disk/network won't compensate for poorly written code
- .. you will have to start caring about all of the above
- .. and learn when to identify boneheaded choices before you make them

Final comments, #2

- .. but don't take it to the extreme!

Comments?

Thanks!

- <http://www.creative.net.au/talks/>