Optimisation and Profiling

Large Scale Programming, 1DL410, autumn 2009
Cons T Åhs
Optimisation

- Your first consideration when writing a program is writing great and beautiful code, not correctness or efficiency.
  - If the code is great and beautiful it can be made correct.
  - Fast but incorrect programs are not so interesting.
- Only when you have a correct program (or part there of) should you consider efficiency.
  - If there are requirements stated for the program they might include non functional requirements stating limits on resource consumption, e.g., performance (time spent on a given task) or memory usage.
  - These might mention what hardware you should be running as well.
  - If possible, the cheapest way to speed up a program is to buy a faster computer (hardware is cheap, developers are expensive).
  - If buying faster hardware isn’t an option, it is time to try to make the program itself be more efficient.
Optimisation

- Do not attempt to write an efficient program from the start!
  - Premature optimisation takes time and will often result in bad code.
  - Also, you are likely to optimise the wrong parts of the code.
  - Experienced developers will tend to write efficient code from the start without actually thinking about it.
- Before starting to optimise you need a clear picture of what to optimise.
- Just looking at the code and guessing where the bottlenecks are will often lead you wrong.
  - Large chunks of code will get attention.
  - Code with known bad complexity will get attention, but it might not really be a problem.
- In the ray tracer we have a clear (and correct) intuition that we will be doing a lot of hit point calculations, but in general this might not be the case.
- You need to find the bottlenecks by careful measuring, refactor the program for efficiency and then iterate the process until you’re satisfied.
  - If you have good tests, the refactoring can be done safely, i.e., it will preserve the existing functionality.

fredag 06 november 2009
Profiling

- Measure, don’t guess!

- Best practices
  - Have a number of different test cases to use for profiling
    - Having too few might lead to optimising special cases.
  - Have test cases of different sizes
    - Get a feel for the complexity of your calculations - lowering the complexity will give a great benefit for larger problems.
  - Keep you test cases constant so you can run them repeatedly and compare different attempts at optimisation - sometimes an attempt will actually make things worse.

- Use tools for profiling
  - Profiling will give you at least time spent in methods and the number of invocations.
  - You can profile memory as well, i.e., see how much memory the program is using and which objects consume space.
Profiling

- Profiling affects the program being profiled and thus takes time so the times given are not “real”. Rather, see them as relative times to be used for finding attack points for your optimisation.

- Two versions of the same program
  - Slow version: 1.94s (normal) vs 40s (profiled)
  - Fast version: 1.26s (normal) vs 21s (profiled)

- Getting real running time must be done externally or by inserting minimally invasive measuring in the code, i.e., obtaining times at a few selected places.

- Profiling samples your code when running, so don’t expect identical times for methods that you “know” should have identical results.
Profiling

› Eclipse - use tptp if it has been installed.
  › eclipse needs to be updated for this, but they are looking into this.
› NetBeans has a profiler built in, so we can use that.
  › Trick:
    › Create a project in NetBeans.
    › Delete the source directory (named src).
    › Create a link to your corresponding source directory in your eclipse project.
  › You can now develop in eclipse and do profiling in NetBeans.
› [show NetBeans]
Optimisation

- Slope style profiles are easy - just go for the largest time spender and see what you can do.
- Flat profiles are more difficult - attacking a single method will not have too much effect. More thinking needed..
- Optimisation is not doing more calculation than necessary.
  - Can you do a calculation later, when you know that you need it?
  - Are there calculations you can avoid totally?
- Changing the algorithm is the most efficient optimisation, i.e., by finding a way to lower the complexity.
  - A lot of thinking might be needed for this.
  - This also makes large changes to the program, so having good tests against interfaces will help a lot.
- Don’t be afraid of throwing code away (at least mentally)
- Understand your problem
- Understand your program
Optimisation

- Low level optimisations:
  - Move invariants out of loops.
  - Don’t calculate the same thing twice.
  - Compute only values that you know you will need
  - Organise conditionals to catch the most common case first

- The total running time of a method is N*Ti; try to lower one or both to make the total go faster.
Optimisation

› In a language such as Java you don’t have too much control over the low level details such as memory allocation, but if you are using a language where you do, there is much to think about.
  ‣ Keep memory access localised to utilise cache.

› The same goes for writing code that makes good utilisation of a processor
  ‣ Again, utilise cache by avoiding jumps.
  ‣ Utilise processor pipelining and prefetch of instructions by keeping a straight flow of the code.