


Complex Components in Mobile Environment



Ing. Daniel Kíř

1



Mobile Environment and Limitations

- High memory complexity
- High computational complexity
- Scale of the problem
- Potential of growth

2

CPUs in mobile devices have lower clocking frequencies and simplified architecture which means that it consumes less power and does well for "simple" problems, it also does poorly in case of complicated problems with high pressure on memory or processing power.

Some problems are acceptable for small data-sets but get fairly prohibitive quite quickly with size of data.

Acceptable solution is no good if it ceases to be acceptable a short while later.

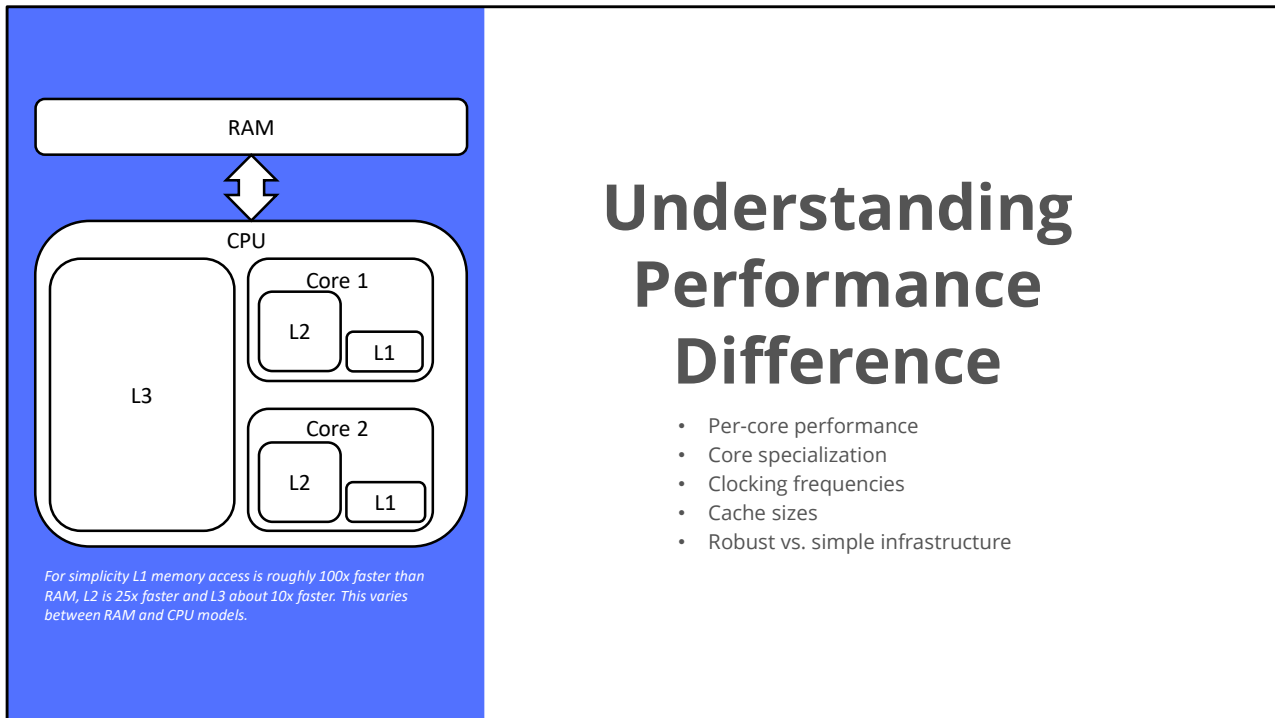
How does this Compare to Server?

- CISC vs. RISC architectures
- Power efficient vs. power intensive
- Scalability



Servers have robust architecture completely dedicated to difficult problems with high pressure on both memory and CPU processing subsystems. This is paid for by power usage.

It is always worth to consider if scaling is needed later on. Adding server to online solution is usually relatively easy while switching from mobile to server in later stages of development is usually quite expensive.



Understanding Performance Difference

- Per-core performance
- Core specialization
- Clocking frequencies
- Cache sizes
- Robust vs. simple infrastructure

- Prime core, performance cores and efficiency cores vs. all high performance cores
- Cache, RAMs on server >> mobile device
- # of cores on server >> mobile device
- Prime core performance vs. server core ~ equal performance
- x86, x64 CISC architecture vs. ARM, ARM64 architecture

How to Identify Problematic Component?

- Product specification
- Feasibility study
- Growth prediction
- Prototyping & Measurements




Identify pieces of software that amount to NP complete or similar complexity problems.

Identify pieces of software which have acceptable complexity but will run on big data sets.

Identify pieces of software which will put enormous amount of stress on data sub-system

Identify pieces of software which have high response time requirements – from these pick those which might have problems with meeting these requirements – these are dangerous in long term

Identify pieces of software which have ambiguous definition or implementation and add definitions/prototypes for them – then see if they fit in any category above



Product Specification: Selected Topics

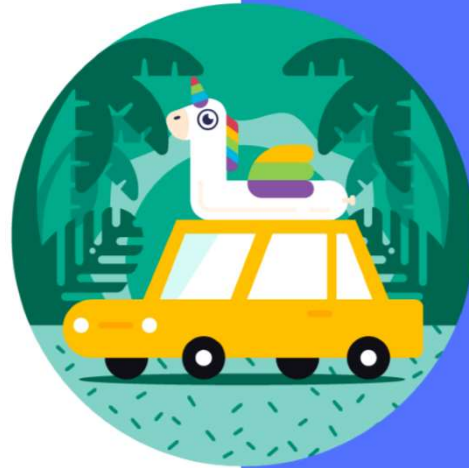
- Target devices
- Performance requirements
- Offline requirement
- Detailed use case description

6

Idea is to define what is expected from software – use cases, constraints, working environment,....

Feasibility Study: Selected Topics

- Primary and alternative approaches
 - Complexity of approaches
 - Pros & Cons
- Scale of problem
- Growth prediction



Idea is to take product specification and produce technological evaluation:

- Can product be produced given product specification?
- What needs to be done?
- Where it can be realized?
- What different options are on the table?
- What are known constraints?
- What are known technological risks?

When to Choose Server?

- Performance requirements
- Power requirements
- Problem scale
- Growth management
- Prototype results



If in doubt that requirements can be fulfilled on device, they usually can't.
Always consider existing server solutions which can be reused (don't re-invent the wheel).

When to Choose Device?

- Connectivity issues
- Offline requirement
- Manageable scale
- Manageable complexity



Usually this is direct consequence of requirements. Some requirements can't be fulfilled by remote service.

There are situations when requirements and reality are mutually exclusive, when in doubt consult stakeholders – bad, unrealizable requirements do exist.

Example: Routing on Road Network

Must be server, right?

Not necessarily. Choose server if:

- Performance requirement
- Precision requirement
- Optimization problem
- Huge data set

Maybe device?

Possibly:

- Offline requirement
- Lower precision
- Basic use case
- Reduced data set

Given properly reduced road network, offline routing is possible. Questions to consider are:

- If some precision issues due to mathematical optimizations are tolerable given requirements
- If highly varying and relatively high algorithm response time is acceptable
- If high power consumption is acceptable for short amount of time

On the other hand server is all powerful routing engine but it also means additional maintenance costs and development costs.

Performance ABC



1

Make sure that selected algorithm is appropriate for mobile devices

2

Consider performance in context of Roofline model (or another benchmark)

3

Measure, adjust, reduce

Try to pick most appropriate algorithm long and short term for actual realization.

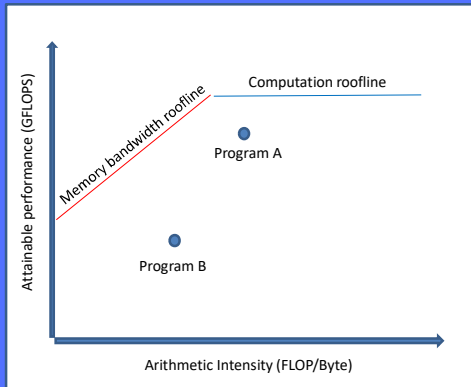
Consider if theoretical performance of algorithm can be achieved on target devices given performance constraints of target devices.

Cross reference this with actual requirements.

Once first version of picked algorithm is developed, profiler and other measurement tools become your best friend.

Properly diagnosing algorithm on device is topic of its own.

Keep in mind that some optimizations for desktop system can have exactly opposite effects on mobile devices (why that is could receive course of its own).



Roofline Model

- Is program running at peak performance?
- Pressure on data subsystem
- Pressure on computational subsystem
- Effectivity vs. amount of work
- Specific for each device
- Differs for memory types

Keep in mind that while simplified chart above is generally correct, it differs for each device and memory bandwidth for each memory type (RAM, L1-3 Cache). One of pitfalls of this model is to pick more efficient algorithm which however does more work so decision will have negative impact.

Pressure on Memory Subsystem?

- RAM & processor caches
- Cache miss
 - Extremely expensive
 - "Waiting processor"



This is again very complicated CPU topic (and could fill entire course by itself). CPU does in a sense have to wait when it finds out that data are not present in its L1 cache – but not really. Waiting is wasteful, typically more robust CPUs will have instructions further down the line ready for out of order execution. It can also apply wide variety of other "trick" such as speculative execution. On memory level it also uses scope of tricks to assure that data are present early. For example CPU can predict what memory segments will be needed in foreseeable future and load them in advance. Desktop versions typically have more robust algorithms to handle this. However, if CPU is really clogged and its memory subsystem is on its limits, executing code will delay significantly. There are techniques how to help your code to be more light-weight on CPU/memory; however, this is again topic of its own.

Can we Design Algorithms for Memory Efficiency?

- Controlling memory access
 - Random access vs. sequential access
- Buffers and lazy operations
- Reducing data scope
- Splitting data and localization
 - Why sometimes making extra copy is a good thing



If you can choose sequential access/strides, it is always better than random memory access (however, keep in mind hash table as negative example, superior algorithm usually wins over better memory ordering).

If you measure that your algorithm is slow due to memory subsystem, consider if hot code segments can be reformed and only apply changes to data if absolutely necessary.

If heavy operation is performed on set of objects and only some objects are used, consider copying affected objects together. This will reduce stress on memory.

Always consider if problem can be split into several smaller ones, this can help in a major way.

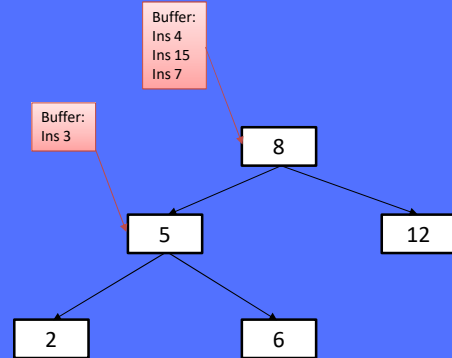
Example: Buffered Tree

Setting

- Get lowest element
- Optimize for insertion
- Memory is slow

Properties

- Average one memory operation per insertion
- Instant access to lowest member
- Expensive Removal



Pressure on Computational Subsystem?

- Algorithmic choice
- Mathematical optimization
 - Heuristic
 - Linear Programming
 - Iterative Methods
- Low Level Optimizations



Sometime software can be slowed down by improper choice of algorithm. This can happen due to poor initial analysis or simply data-set have outgrow initial implementation.

If algorithm cannot be changed or best algorithm is already in place, always consider removing parts of solution artificially. For example if we can remove part of data set at any point because we can prove that final solution is not present there, then remove it.

When both methods above are spent, then try to optimize your code. To mention few techniques – reserving data sets in advance with appropriate size (reduces work with heap), utilizing auto-vectorization, considering branching of algorithm....

Example: Routing on Road Network

Setting

- 80M interconnected roads in Europe
- Calculate route from A to B with air distance 500km

Techniques

Heuristic: Disregard unpaved roads
Linear programming: Consider only roads located within ellipse defined by A and B as its focal points
Algorithm: A*



Do we Face same Challenges on Server?

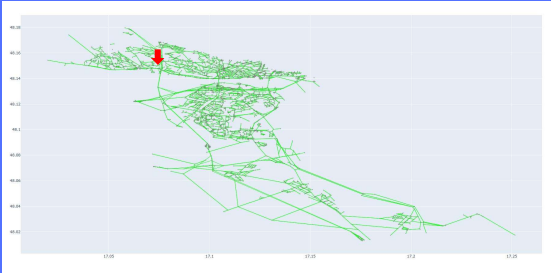
- Moving from "How to fit problem" to "How to make conventional algorithms faster"
- Problems that don't fit on mobile devices
- Wider CISC toolset
- Performance superiority



Many problems which are appropriate for server simply need more parallelism or more memory – this server can delivery (however there are exceptions – for example cryptographic problems). Some algorithms are same but in bigger and more powerful environment, some modify data sets, some have same data sets but algorithm is completely different.

Example: A* vs Contraction Hierarchies

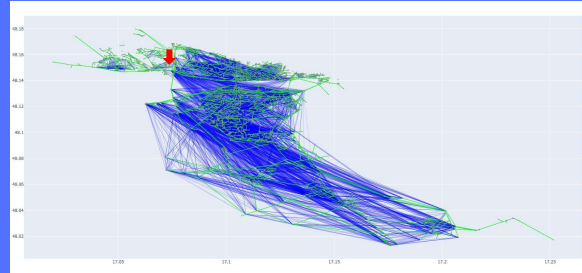
A*



Slovakia

- Road network size - 10MB
- Query average time - 200ms

Contraction Hierarchies



Slovakia

- Road network size - 1GB
- Average query time - 0.5ms

So, how do we Approach this?

- Systematic approach to a meaningful software decision
- Documents, documents and more documents
 - Product specification
 - Requirement analysis
 - Feasibility study
 - Architectural documents
 - Measurements, prototypes
 - Documented educated guesses



20

So, at the end of the story, each case is different, there is no one size fits all approach to development of complicated subsystems. All we can do is follow systematic approach and solution will eventually firmly fit on server or device. Keep in mind that methodology is only as good as people who follow it. Documents mentioned above try to make sure that all stakeholders are involved in process and have their say. Project usually fail due to failed communication or information channels.



Biggest Hazzard is Ambiguous Requirement

Definition: Requirement

- 1) A condition or a capability by a user to solve a problem or achieve an objective.
- 2) A condition or capability that must be met or possessed by a system or a system component to satisfy o contract, standard, specification, or other formally imposed documents.
- 3) A documented representation of a condition or capability as in 1) or 2).

[IEEE 610.12-1990]

How do I Identify well formed Requirement?

Requirement:

- No wishful thinking
- Places resolute unambiguous constraint on product
- Can be realized by algorithm or strategy
- Unambiguous acceptance criteria/tests are part of requirement

Note:

- There are multiple strategies for requirement elicitation



22

To a degree this is an engineering art and experiences play a major role. We try to evaluate requirements in terms of their content and guess if they can serve as a meaningful source of information for software development. Good questions to ask are:

- Can I create architecture based on this?
- Can I create software based on this?
- Can I create acceptance criteria based on this to verify that the requirement has been fulfilled?

If the answer for any question above is "no", then the requirement is incomplete or outright bad.

Example: Good vs. Bad Requirement

Bad: Make good and fast routing.

- How do I measure "good"?
- How fast is "fast"?
- How do I validate this?
- Make where?
-

Good: Make routing for CAT 4s mobile device which will route standard car BA-KE through D1 under 100ms without internet connection using Sygic maps.

- Much better.
- Still can be improved upon.
- Maybe sub-requirements?
- Proper glossary?
-

Requirement engineering have formalized methods to verify integrity of requirements. Requirements in so called "natural language" are especially tricky.

Why Bother with Architecture?

- Details matter
 - Danger of developing unusable software
 - Danger of imposing artificial constraints
- Different teams can review software before it has been developed
- Saves time in later stages of development in all but most simplistic systems



24

Small changes in architectural model can amount to huge changes in code base therefore it stands to reason that prior modeling and model verification is beneficial for project and saves resources.

It is common and quite natural to consider oneself immune to major architectural error; however, this is always illusion. Major changes do occur in software all the time no matter how excellent the architect is. Modeling is simply cheaper way to both detect segments which needs to be redone and validate software as whole.

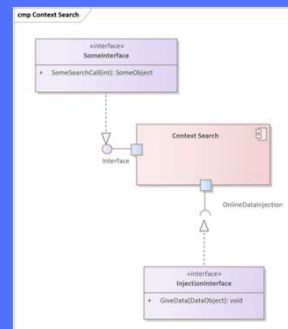
Example: With & Without Architecture

Without architecture

A: We developed context search. [600h]
B: Can we inject online data?
A: Oops. [300h]
Grand total of 900h.

With architecture

A: We modelled context search. [30h]
B: Can we inject online data?
A: Oops. [3h]
...
A: All done and committed. [600h]
B: Well done.
Grand total of 633h.



Closing Remarks



- Creating proper initial documentation, tests and specifications can be boring; however, spending two months fixing software after deadline is even more bothersome.
- Plan, code, measure and repeat.
- Be open to new ideas, technologies and opinions; open minds tend to open locked doors.

Questions?

