Optimizing Browsing Experience

Rodolph Perfetta

ARM

The Challenge

- ARM based devices can offer
  - Better battery life
  - Slimmer form factor
  - Lower cost
- But
  - Software is primarily written for desktop platforms
  - Need to be optimized for ARM and mobile environment

Agenda

- Javascript Engine - Today
- Javascript Engine - Tomorrow
- Optimizations and Architecture
- Memory and Power
- ARM CPU and Memory Architecture
- Profiling
- Software limitations

JavaScript Engine - Today

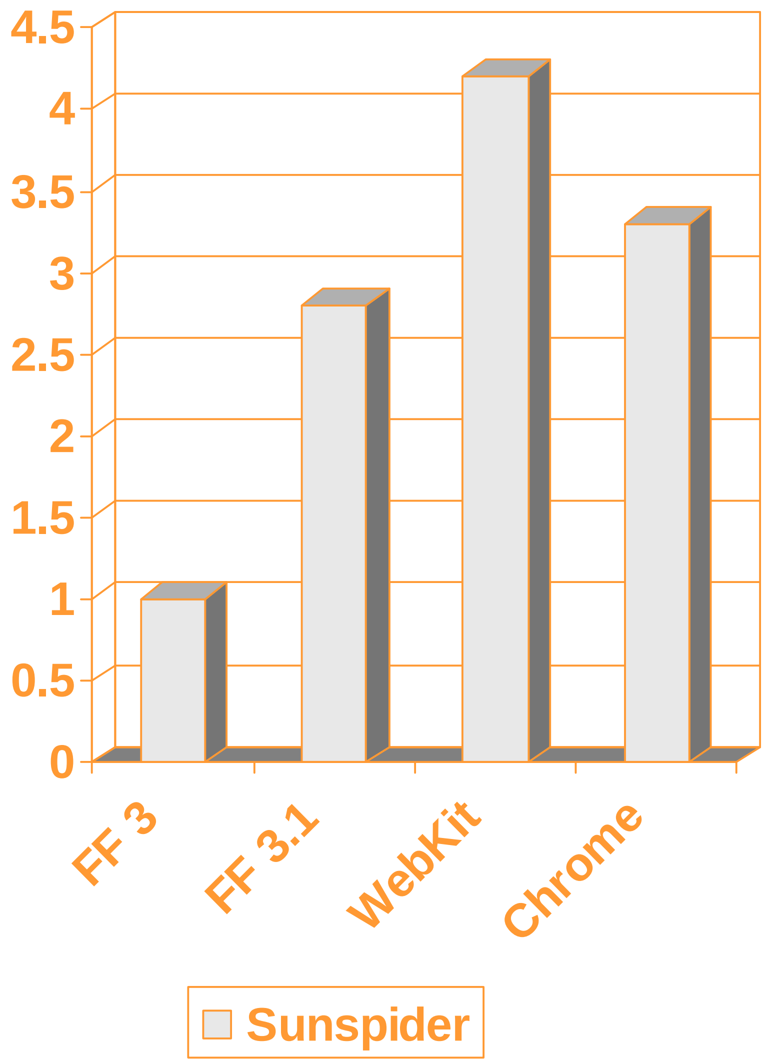
- Javascript VM are in their infancy when it comes to performance optimizations
- FireFox 3, Safari 3, IE7, Opera
  - Bytecode interpreter
  - Syntax tree walker
- Chrome 1
  - JIT to native code
- While optimizations are nice to have on powerful hardware, they are critical on small devices

JavaScript Engine - Tomorrow

- Google Chrome
  - All the code is compiled before being run
- Safari 4.0 Beta
  - Mixed interpreted/compiled code
  - Use JIT for RegExp as well
- FireFox 3.1 Beta
  - Interpret, trace then compile
  - Aggressive type specialisation
- There are more than one way to improve performance
  - some ways are more “embedded friendly”
  - Think of memory
  - Think of power

JavaScript Engine - Tomorrow

- SunSpider benchmark
- Performance normalised to FireFox 3
- Higher is better



Opt: Polymorphic Inline Cache

- Used by V8 (Chrome) and SquirrelFishExtreme (Safari 4)
- Assign “types” to object based on their fields definition
- On field access:
  - Cache first used offset and corresponding type
  - If the next field access is on the same type of object then reused cached offset
  - If not redo lookup and update caches
- Implementation
  - Offset and type are in-lined in the code
  - Every cache miss will trigger a rewrite of native code
  - On ARM this will require a cache sync
  - Cache sync operation is not free, if the cache hit rate is not high it won’t be worth the effort.

Opt: Tracing and Type Specialisation

- Used by TraceMonkey (FireFox 3.1)
  - Interpret the code
  - Record exact types used
  - When a loop is detected generate code with gathered type information
  - If types change, retrace, recompile
  - Code is generated in buffer of 4k, no requirements for the buffer to be contiguous
- Performance can already reach non optimized C code
- Potential to be memory friendly:
  - Only compile what is run
  - chunks can be deallocated

Memory and Power, Power and Memory

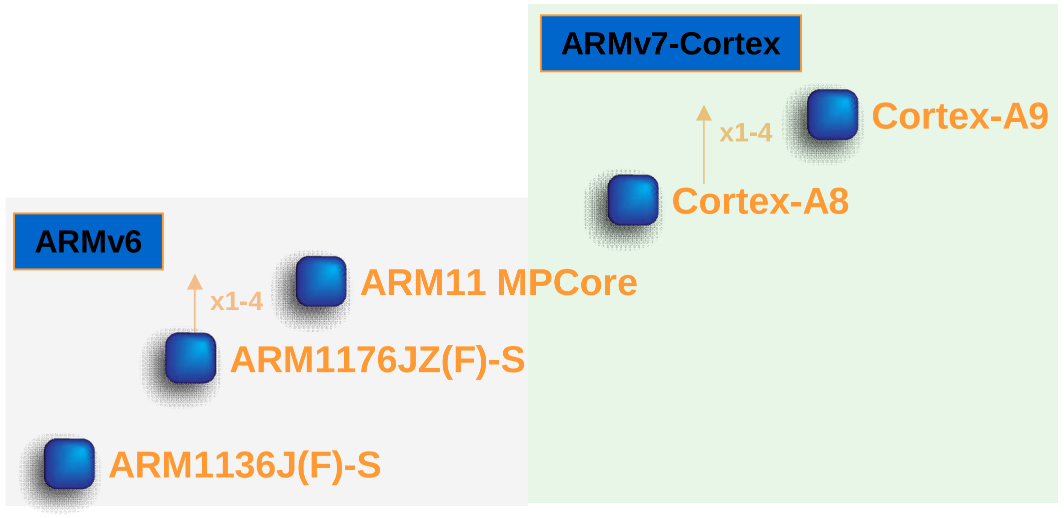
- Memory is key to performance on ARM systems
  - Limited amount: Nokia N810 has 128M
- Less memory accessed/used usually means:
  - faster applications
  - cheaper devices
  - better battery life
- Unused CPU cycle are not free on a mobile device
  - While idle an ARM CPU will consume 200-300 times less power
  - Keeping the CPU alive is costly
- The faster your task run the more battery you get

Memory Footprint

- Do you know how much memory your application uses?
- Code footprint
- Data footprint
- OS instrumentation
  - patch to the linux kernel
  - Dynamic instrumentations
- Malloc/new instrumentation
  - simpler
  - good enough to start with
- There are multiple malloc libraries out there

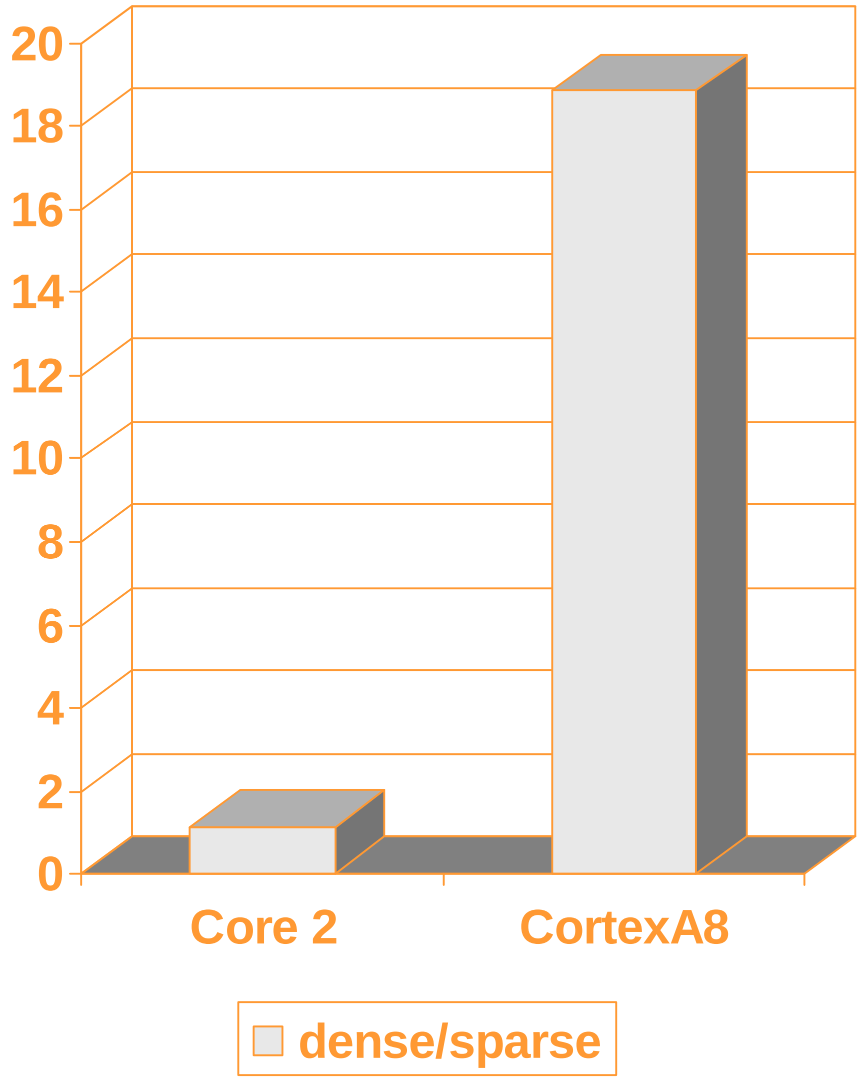
ARM CPU and Memory Architecture

- Outside L1/L2 caches it is different for every licensee
- Do not assume anything about RAM (e.g. latencies)
- Instruction and Data cache need to be managed manually
- CortexA have L2



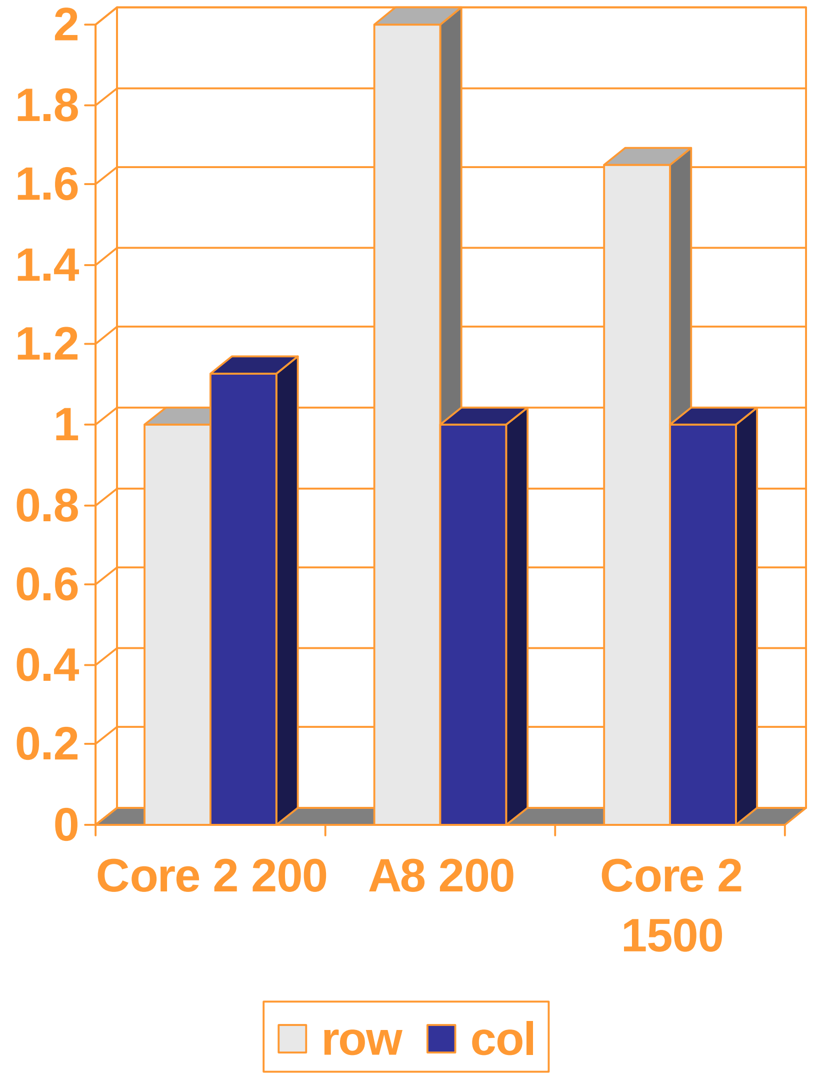
On Target Profiling - Scaling down

- Profiling/tuning on a different architecture has little value
- E.g looking up a 32bits key in a table
  - Sparse (key every 8 words)
  - Dense
- Searching 30000 keys
  - Macbook (Core 2 Duo)
  - CortexA8 board
- Dense speed up vs sparse



Off Target Profiling - Scaling up

- Your target may not always be available
- Try scaling up on a more powerful platform
  - Will not give you a realistic profile
  - Can help spotting potential issues
- Ex matrix multiplication
  - Row first
  - Column first
- Speed difference with various matrix sizes
  - Macbook (Core 2 Duo)
  - CortexA8 board



Choose the right ISA

- ARM: Full speed, Full size
- Thumb: 20% smaller, 20% slower
  - Thumb was design as a static compiler target
  - Thumb is useful for system with 16bits memory
  - Thumb is not JIT friendly (non consistent constraints, small range for branches, less registers etc)
- Thumb2: Full speed, 20% smaller
  - Thumb2 addresses Thumb’s issues
- ARM is still the most flexible of the three ISA

Use the Hardware

- VFP is becoming more available, make use of it
  - Available on some ARMv6 (ARM11)
  - Available on all ARMv7A (CortexA8, CortexA9)
- Thumb2 can make a difference for big code base
- Neon (128bits SIMD)
  - Useful for codec
  - gcc mainline does not generate Neon code
  - gcc mainline support Neon assembler
  - If you want it, do it yourself

Know the software: Procedure Call Standard

- When passing parameters
  - first four word sized parameters in register 0 to 3
  - The rest on the stack
- Sub word sized parameter take a full register
- 64bits values are passed in an even + consecutive odd register. On the stack 64bits values are 8 bytes aligned
  - f1(int a, double b, int c): a->r0, b->r2+r3, c-> stack
  - f2(double a, int b, int c): a->r0+r1, b->r2, c->r3
- Avoiding parameters on the stack usually result in smaller and faster code
- GCC has not yet implemented use VFP register for passing parameter
- More info at http://infocenter.arm.com

Summary

- When doing mobile software development
  - Keep in mind memory, power
  - Use the hardware fully
  - Profile on each target
  - Know the surrounding software limitations