Talk #2:

Tales of a Lua embedder thrown into the JavaScript world

Eric Wing
@ewingfighter
http://playcontrol.net

#JavaScript #Lua
#JavaScriptCore #v8
#VM embedding #NoWebBrowser
#Android NDK #iOS
#cross-platform
#videogame dev #game engine
#Appcelerator #BlurrrSDK

Background: Lots of different hats to wear

- Worked on Globalstar (orbital satellite global communication system)
- Cross-platform: Linux, Mac, Windows, iOS, Android, FreeBSD, Solaris, IRIX
- Scientific Visualization (OpenGL)
- Native Cocoa Application development (Mac, iOS)
- Video Game Engines
- Open Source: SDL, OpenSceneGraph, CMake, OpenAL
- C, C++, Obj-C, Lua, Perl, Java, JavaScript, Swift
- Book author (Beginning iPhone Games Development)
- Startup cofounder

Disclaimer

- This talk is based on things that happened several years ago
  - Some info may be old
- We are building libraries for developing consumer facing apps
  - Things like shipping binaries and stable ABIs are really important
- Don’t have time to cover language level differences between JavaScript and Lua
  - Though many surprising accidental similarities
  - See Roberto Ierusalimschy: Lua versus Javascript: Why do we need multiple languages?
    - http://www.inf.puc-rio.br/~roberto/talks/www2013.pdf

Appcelerator

- Makes Titanium SDK
  - Write native apps in JavaScript for iOS & Android
    - Provides cross-platform API in JavaScript
  - Embeds JavaScript engine like we would with Lua
  - No web browser is involved

Grow the market

- Appcelerator builds customer base in enterprise and traditional app developers
    - Wants to try to get video game developers
        - Want a library module that Titanium users can hook into if they want to do games

- This is where my story begins…

Platino

- Funded by Appcelerator + big giant company
- Co-founder already built usable prototype using official Titanium APIs for creating 3rd party libraries
- Good enough to close deal, but serious performance bottlenecks and limitations in real world use
- Titanium language bridging is really slow
- Abstractions so thick, hides two different JavaScript VMs as implementation details
  - iOS used JavaScriptCore
  - Android used v8
- Titanium doesn’t allow for single cross-platform modules written in C
  - Must write separate Obj-C and Java versions
- This is a non-starter…

Deep Breath

- We need a deep understanding of the system architecture
- While we can see the bridge crossing is “slow”, we haven’t nailed down solid numbers
    - Nor do we know our theoretical maximum

The Plan

- Hack & bypass Titanium's layer and talk directly to JavaScriptCore & v8
- Measure bridge crossing performance of Titanium vs. direct access
- For context, also measure Lua and compare

	Titanium Android v8	Titanium iOS JSCore	Direct Android v8	Direct iOS JSCore	Direct Android Lua	Direct iOS Lua
pass double from JS to native 10 million timess	41.4 sec	596.4 sec (* extrapolated)	6.2 sec	N/A (bug)	2.2 sec	3.0 sec

- Android is Nexus 7
- iOS is iPad mini
- Nexus 7 is faster hardware than iPad mini
- Technical issue with the Titanium/iOS side made direct comparisons tricky. Ultimately from other things, we concluded Direct JSCore is in the ballpark of Direct v8.

Conclusions

- Titanium bridge was several orders too slow
- Direct JS was ~2x-3x Lua
- Lua was expected to win (hard to beat it)
- Overall Direct JS performance acceptable
- Now have good mental context of what we can do

Phase 2: How do we make our libraries?

- JavaScript does not have a standardized API for talking to C
  - v8 and JSCore require completely different binding code
  - Huge burden for us to support
  - Both APIs not very well documented or used
    - Hard to get help (mailing lists not very responsive)

SWIG to the rescue

- Grand-daddy binding generator for ~20 years
- Experimental v8 & JSCore generators in progress
- Still lots of missing features, bugs, and work for us
  - But still very welcome over the alternative

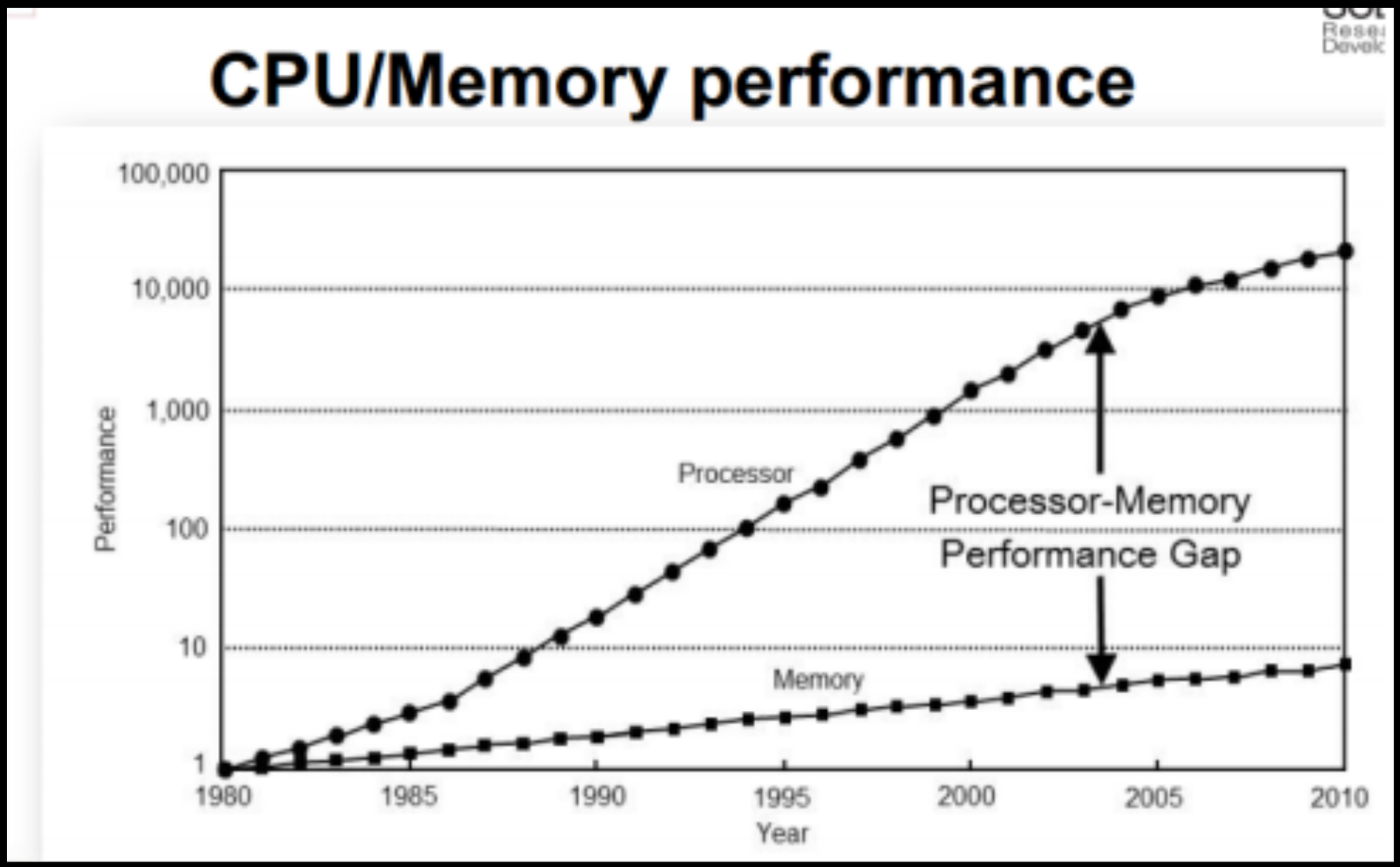
But JavaScript Culture…

- (Too much?) Faith in "*fast*" JavaScript VMs
  - JIT and compilers solve everything
    - "Google v8 is the best in the world with the smartest people in the world"
- Can't we just use a pure JavaScript implementation?
  - Skeptical, but we should test/prove

Culture Clash:
JS vs. native game dev

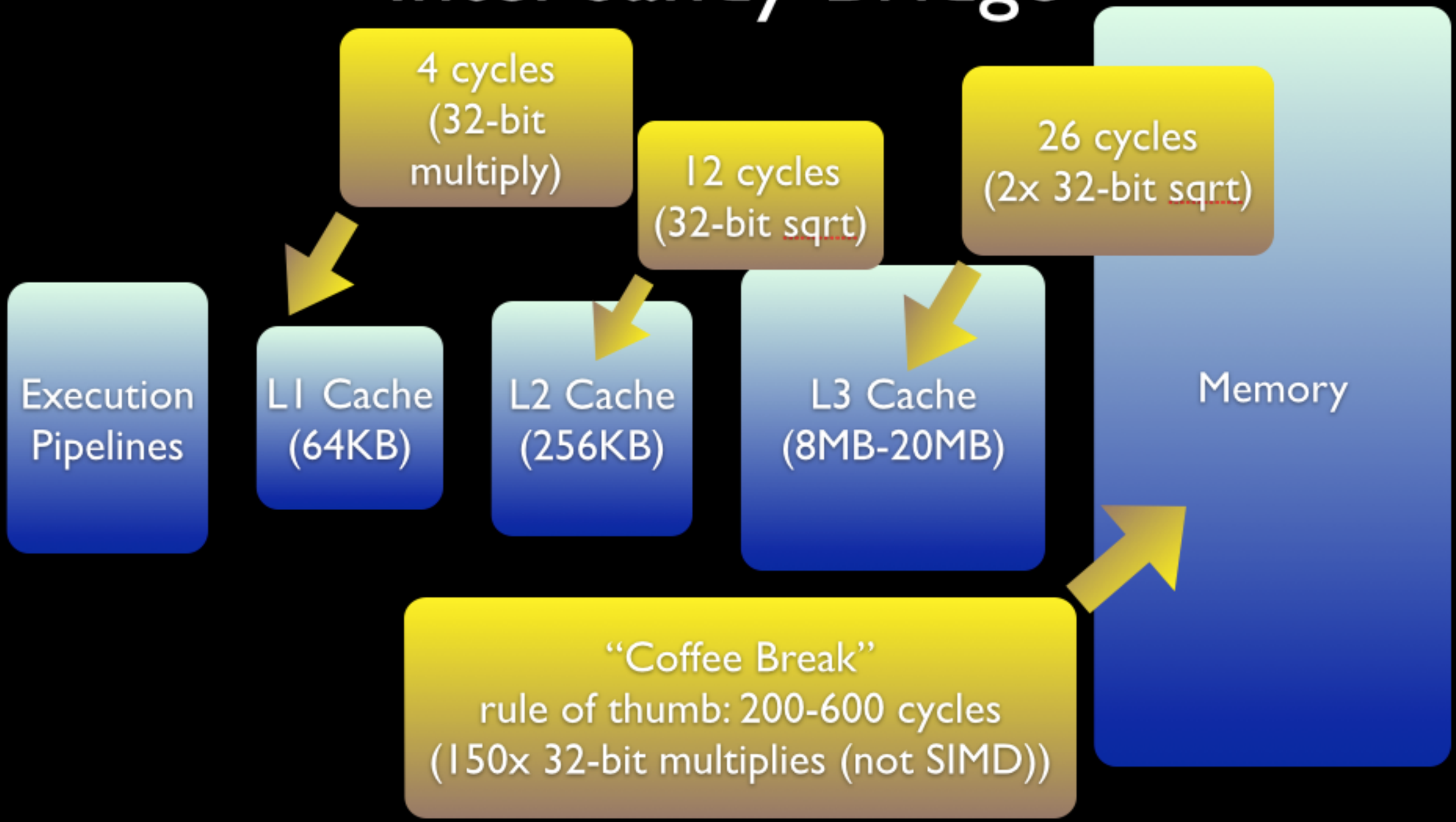
- JS: “Native code is fast. And JIT creates native code.”
    - Game Dev: “What is the hardware doing? L2 cache miss?”
- JS: “Multiple threads and async make things fast.”
    - Game Dev: “Stop interrupting/blocking my game loop. I ***must*** complete this in 16 ms.”

People forget Memory Access is I/O
Memory is a major bottleneck too



- http://research.scee.net/files/presentations/gcapaustralia09/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf

Intel Sandy Bridge

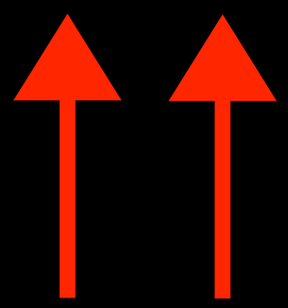


- Source: Eric Brumer, Native Code Performance and Memory: The Elephant in the CPU, //build/ 2013

L1, L2, Main RAM access times compared, ***to scale***
Mike Acton (CppCon 14): Data Oriented Design & C++

Back in (pure) JavaScript…
(or why we bother with native)

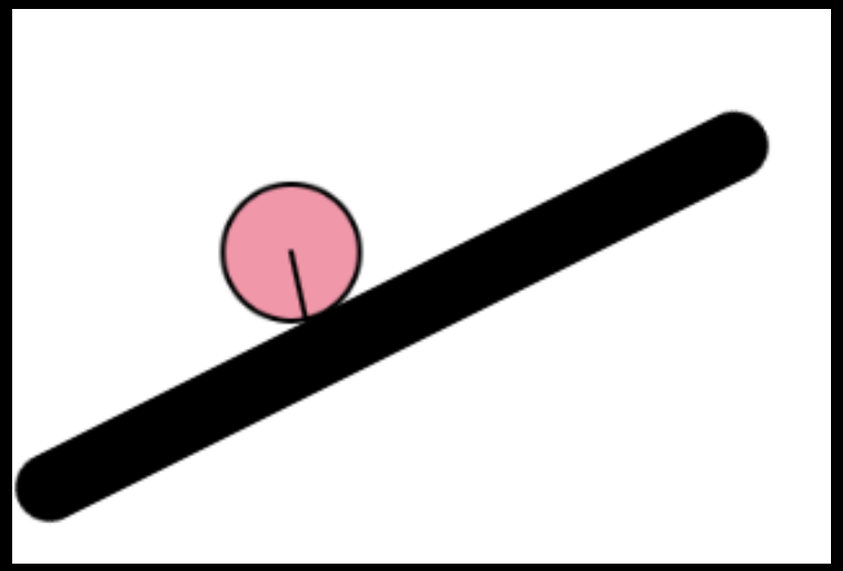
- foo.bar.baz();



hash: O(1), but slow

- function Vec(x, y) { this.x = x; this.y = y;}
  - What's the memory layout?
- "Compilers are a tool, not a magic wand"
  - Can only reason about 10% of your code

Benchmark of a physics library



- Is “pure JavaScript” as fast as native?
  - (Or close enough?)
- This benchmark is imperfect, but good enough for general magnitudes

	JavaScript	Native (C/JS bridge)	x faster
iPad mini (JSCore no-JIT)	208 ms	36 ms	5.8x
Samsung Galaxy S4 (v8+JIT)	538 ms	13 ms	41x
iOS Simulator (JSCore no-JIT)	28 ms	4 ms	7x

Observations of the results

- Forgot to compile iOS native with optimizations...it's probably even faster
- The S4 is more powerful than the iPad mini in terms of hardware
  - The native vs. native performance was 2.8x faster on the S4 than iPad mini
  - But the JavaScript vs. JavaScript performance was 2.5x **slower** on the S4
    - And v8 is ***with*** JIT vs. JSCore with **no**-JIT
- So => JavaScriptCore with **no**-JIT *on slower hardware* beats v8 ***with*** JIT *on faster hardware* by over 2x

Conclusions

- Google v8 hype didn’t live up
- JIT hype didn’t live up
    - (lots of reasons why JIT on mobile may not be impressive)
- C is faster

Fight Club

- What is the First Rule of ***Library*** Programming?
  - You do not use C++
- What is the Second Rule of Library Programming?
  - ?

An experience with Google v8…

- But first talking about the Android NDK will help

Android NDK (**N**ative **D**evelopment **K**it)

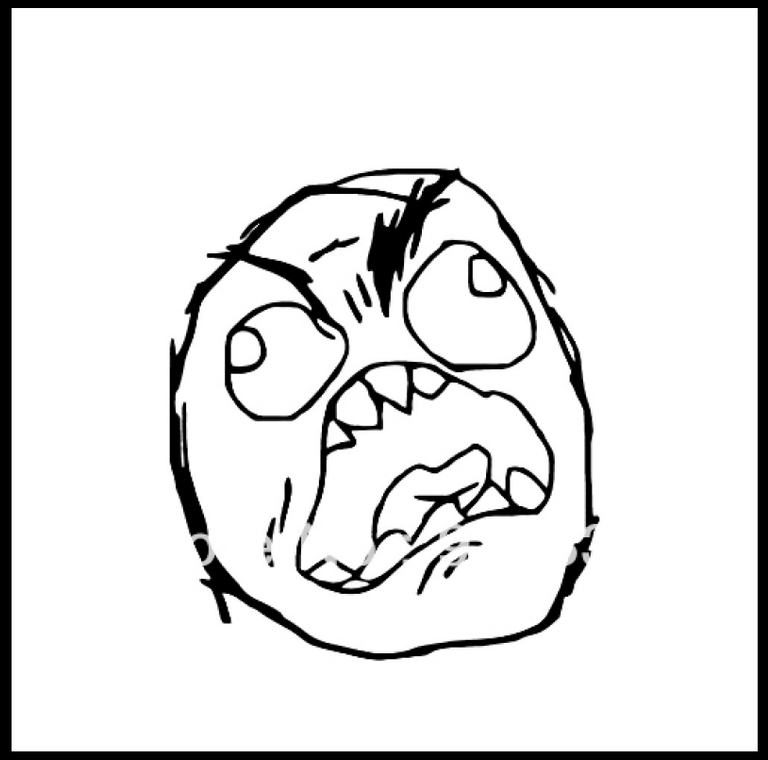
- All Android apps ***must*** be written using the Android **S**DK which is in Java

- Android NDK provides gcc & clang so you can create dynamic libraries in native languages like C & C++

  - (Since our JavaScript engines are written in C++ and have C/C++ APIs, these live in the NDK side)

- Your Android app must use Java's LoadLibrary to load the dynamic libraries, and then communicate using JNI

Android NDK is a miserable experience



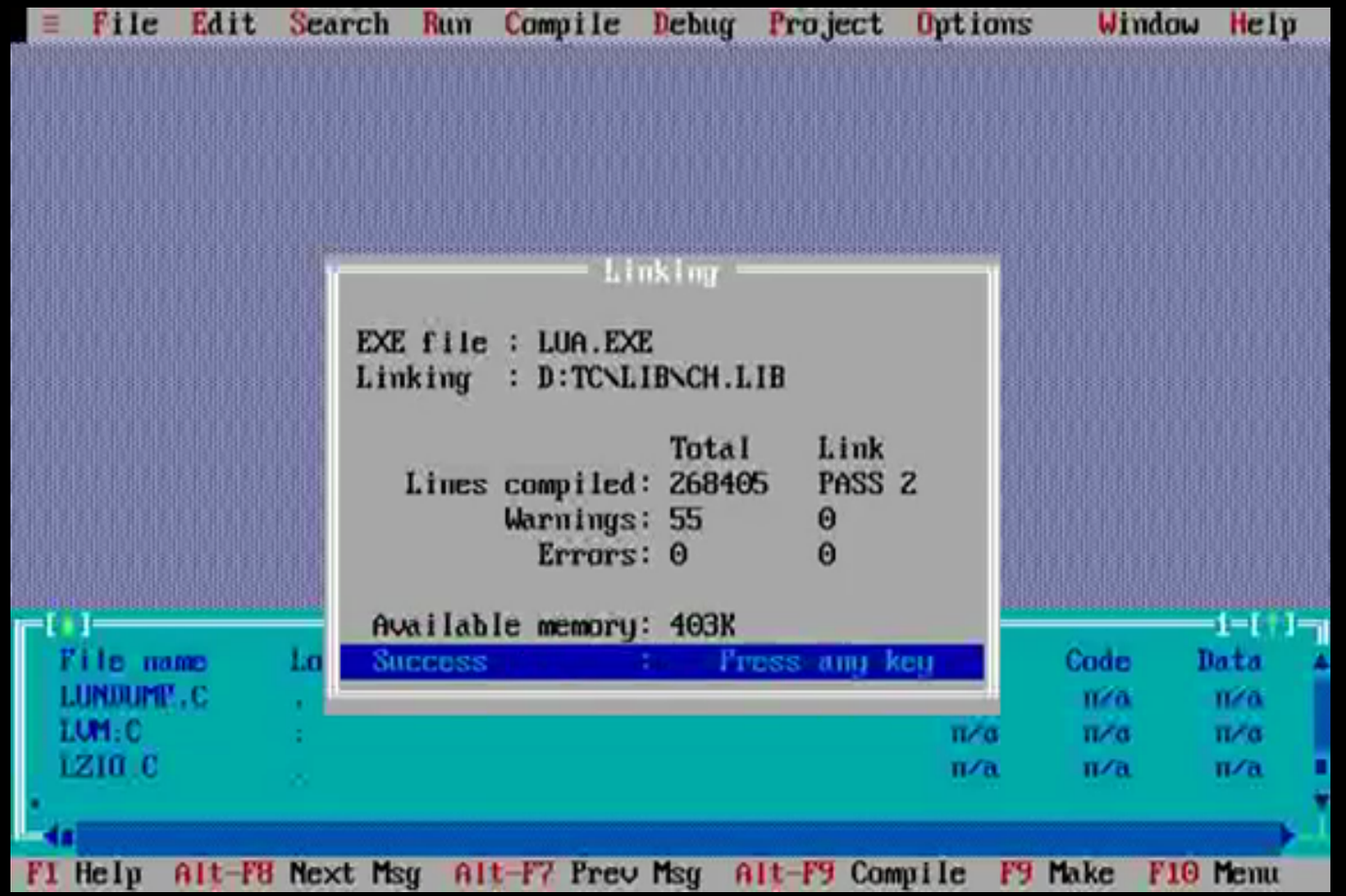
- John Carmack - “Half baked”
- Second class citizen on Android
- IDE and build systems not well integrated
- Almost no Android libraries are provided in the NDK
- Lots of things are broken, slow to get fixed, if ever
- Word on the street (as of then): Only 2 full-time Google engineers + a few part time
    - Consistent with number of Google employees on NDK mailing list
    - No slight intended on those 2 engineers. Valiant effort. Google treats them as the black sheep.
        - Google: Among the richest, powerful companies in the world with #1 dominance in mobile, and this is the best effort Google chooses to put in

Bionic (the C library)

- Android does not use glibc (like *real* Linux)
- Wrote their own called “Bionic”
- Doesn’t care about POSIX compliance
- Doesn’t even care about ANSI compliance
- 8 years into Android, still terrible

Lua 5.2 (2013) builds with Turbo C 1.0 (1990) for MS-DOS without modification

https://youtu.be/-jvLY5pUwic



Android NDK (r11c) fails to compile Lua

```
Building armeabi
Scanning dependencies of target lua_library_static
[  1%] Building C object CMakeFiles/lua_library_static.dir/src/lauxlib.c.o
[  2%] Building C object CMakeFiles/lua_library_static.dir/src/lbaselib.c.o
[  4%] Building C object CMakeFiles/lua_library_static.dir/src/lbitlib.c.o
[  5%] Building C object CMakeFiles/lua_library_static.dir/src/lcorolib.c.o
[  6%] Building C object CMakeFiles/lua_library_static.dir/src/ldblib.c.o
[  8%] Building C object CMakeFiles/lua_library_static.dir/src/linit.c.o
[  9%] Building C object CMakeFiles/lua_library_static.dir/src/liolib.c.o
In file included from /Users/ewing/Source/Blurrr/Lua/lua53hack/src/lua.h:16:0,
                 from /Users/ewing/Source/Blurrr/Lua/lua53hack/src/liolib.c:20:
/Users/ewing/Source/Blurrr/Lua/lua53hack/src/liolib.c: In function 'read_number':
/Users/ewing/Source/Blurrr/Lua/lua53hack/src/luaconf.h:640:47: error: 'struct lconv'
 has no member named 'decimal_point'
 #define lua_getlocaledecpoint()  (localeconv()->decimal_point[0])
                                               ^
/Users/ewing/Source/Blurrr/Lua/lua53hack/src/liolib.c:424:13: note: in expansion of
macro 'lua_getlocaledecpoint'
   decp[0] = lua_getlocaledecpoint();  /* get decimal point from locale */
             ^
make[2]: *** [CMakeFiles/lua_library_static.dir/src/liolib.c.o] Error 1
make[1]: *** [CMakeFiles/lua_library_static.dir/all] Error 2
make: *** [all] Error 2
```

Insights into Bionic

- **To**: tech-userlevel%netbsd.org@localhost
- **Subject**: **getting Android's bionic C library back in sync with upstream**
- **From**: **enh <enh%google.com@localhost>**
- Date: Wed, 25 Jul 2012 15:14:49 -0700

---

hi, my name's Elliott, and i'm a C library maintainer.

as you may be aware, Android's C library "bionic" is a hybrid of code from different sources --- there's stuff like the pthreads implementation that's home grown, but there's a lot of FreeBSD, NetBSD, and OpenBSD code in there too. NetBSD files aren't from a single NetBSD release; hell, they're not even necessarily from _any_ release. i found a bug in getcwd (iirc) that had been fixed upstream in like 1996, but we had some random old version of that file. (and similar for the other BSD flavors.)

i'd like to get bionic to a point where we have one upstream (that is, i'd like to get all my upstream code from FreeBSD _or_ NetBSD _or_ OpenBSD, not a mixture), and where we can easily track that upstream. so i have a few questions:

https://mail-index.netbsd.org/tech-userlevel/2012/07/25/msg006571.html

Terrible performance bug for strlcpy

- Wrote a test program to run Test262 suite on JavaScriptCore
  - Runs through 11,000+ files
  - Used strlcpy
- Mac, iOS so fast that I didn’t think about it
- Android took: 9000 ms
  - Switch to strncpy + manual NULL term: 14 ms
- Maybe I should be impressed they provide this function at all

Android file system and the .apk

- Files that ship with your app are inside the ".apk" (think .zip)
  - Can't use standard C file family (fopen, fread)
  - AAsset* AAssetManager_open(AAssetManager *mgr, const char *filename, int mode);
    - Needs a "God" object from the Java Android Activity or Context class
  - Existing cross-platform (ANSI) C/C++ libraries won't work without modification
    - This includes the Lua I/O library
- Places outside the .apk can use C file family
  - "Internal Storage", but may not have much storage space
  - "External Storage", but may not exist or have correct access permissions

Android SDK/Java isn't that much better

- Get a list of files ('ls') in a directory of an APK is well known to have a serious performance bug
- Get list 11,000+ files: I killed the process after 3 hours of waiting

Dynamic Library System wonky too

- System.LoadLibrary doesn't automatically load dependencies
  - Load must be manually done & in the correct order
  - Will silently ignore if a library by the same name is already loaded
- Android extracts out your .so files and puts to /data/data/packagename/lib/
  - But this is private implementation detail that can change with no API to know where the libraries are actually put
  - Libraries must always be prefixed with 'lib'
  - Never use soname versioning

Example Impact: Lua module system

- LuaSocket has multiple “core.so” files in different subdirectories by convention
- You can’t have subdirectories
  - (you have no control where the .so’s are placed)
    - This also means file names must be unique or they overwrite each other
- You must have a lib prefix: “libcore.so”
- Solution:
  - Turn off Lua ‘treat dots as subdirectory’ (CSUBSEP)
  - Name files like libsocket.core.so, libunix.core.so

iOS not off the hook

- 3rd party dynamic libraries forbidden (until recently)
- End up statically linking everything
- To preserve normal Lua semantics
  - Modify Lua to use dlsym with RTLD_MAIN_ONLY to find symbols inside to preserve module system semantics
  - Create a sensible module directory layout inside your app bundle for LUA_PATH/LUA_CPATH

Android NDK and C++

- Android NDK provides ***5*** different C++ standard libraries you have to choose from
  - libstdc++, gabi++, stlport, gnustl, libc++
  - All are incompatible with each other
    - NDK API level doesn't help either
- C++ standard library does not guarantee a stable ABI so every NDK update potentially breaks
- If you dynamically link
  - You must bundle with your app because Android does not ship one with the OS
    - In contrast Apple, ships system wide and tries to keep the ABI stable/ backwards compatible

Android NDK and C++

- In the real world
  - People build binaries of libraries and share them
  - People don’t upgrade NDKs at the same time and versions get mixed
  - People use multiple libraries, all built under different NDK versions
- So the final application must include a copy all these different C++ library versions
- But Android doesn’t name versions differently
  - So files overwrite each other
    - And bad things can happen
  - In contrast, Microsoft Visual Studio at least has the sense to put the version number in the file name (e.g. msvcp140.dll)

Android NDK and C++

- So we should statically link, right?
  - Errr…

Static runtimes

Linking the static library variant of a C++ runtime to more than one binary may result in unexpected behavior. For example, you may experience:

- Memory allocated in one library, and freed in the other, causing memory leakage or heap corruption.
- Exceptions raised in `libfoo.so` going uncaught in `libbar.so`, causing your app to crash.
- Buffering of `std::cout` not working properly

In addition, if you link two shared libraries–or a shared library and an executable– against the same static runtime, the final binary image of each shared library includes a copy of the runtime's code. Having multiple instances of runtime code is problematic because of duplication of certain global variables that the runtime uses or provides internally.

This problem does not apply to a project comprising a single shared library. For example, you can link against `stlport_static`, and expect your app to behave correctly. If your project requires several shared library modules, we recommend that you use the shared library variant of your C++ runtime.

Android NDK and C++

- So Lose, Lose
  - Thanks (for nothing) Google
- In practice, I personally found static linking to work better

So back to v8…

- v8 public API is C++
  - Uses templates and classes
  - The API is allowed to break any time
- Titanium uses v8, thus Titanium must use C++ and it spreads all over
- Titanium dynamically links to (an older) gnustl
- Titanium build process ships their gnustl so apps will get it
- Platino grabs the NDK from Google (different time than Ti) for our (pure) C libraries
- Platino must also build against Titanium APIs and v8, so we now have a C++ dependency we didn't really want or expect
- But now we have a problem because our versions don't match
- We didn't notice this
  - Got weird, hard to reproduce crashes. Spent days. Traces made no sense
  - Got lucky and thought about this scenario because there weren't any good clues

Fight Club

- What is the Second Rule of Library Programming
  - You do not use C++

Appcelerator Hyperloop

- Titanium major redesign to address many short-comings we helped discover
- Build a compiler that will read JavaScript code and automatically generate and compile bindings when calling a native platform (Cocoa/Android) API
    - Fixes Titanium bridge overhead problem
    - Allows AppC to implement native Cocoa/Android features in JavaScript instead of Obj-C/Java
- Really cool concept: Lua community should consider doing something like this
        - (Also check out RubyMotion though which predates, but even more ambitious)
- Trade-offs
    - Compile process required (more waiting, live coding harder)
    - Potential binary bloat (static bindings coverage for Cocoa/Android APIs is huge)
        - Indirect and inter-dependencies require more binding code than you may think
- Just went 1.0 a few months ago (open source)

Redesign: Unify around a single JavaScript engine

- **JavaScriptCore** chosen
  - Built-in on iOS
  - Though C++ implementation, pure C public API
    - Helps contain from spreading
  - Apple has kept API stable forever (remember my other talk and JSCocoa?)
- **v8** cannot disable JIT (disallowed iOS, WinRT)
  - Always breaks API
- Mozilla **SpiderMonkey** considered due to reputation
  - C++ API
  - not standard on any of the platforms we care about
  - No SWIG (for Platino)
- **Rhino** (Java based VM) was used a long time ago by AppC for Android and was terrible
  - Dalvik performance was always awful
  - Wouldn't work on iOS due to Java
- I personally like **Duktape**, but not one of the "big" ones
  - "Nobody ever got fired for buying IBM"

Duktape: JavaScript VM designed from the ideas of Lua

- ANSI C (no C++)
- Small, minimal dependencies
- Stack based C API
- Bytecode
- Simple, No JIT
- Experimental SWIG generator contributed just last month

Code and RAM footprint

	x86 default	x86 lowmem	x86 full lowmem
Code	170kB	150kB	200kB
Startup RAM	68kB	35kB	4kB

Porting JavaScriptCore to Android

- WebKit is ~5GB Git repo download
- Platform specific code
- Extremely complicated code base
- Extremely complicated build system
  - (Intertwined with WebKit)

Porting JavaScriptCore to Android

- Initial attempt by somebody else to implement new build system with Google Gyp
- Simplify: Try to reproduce Mac Xcode build first to see if the Gyp is right
- ~2 weeks, couldn’t get it working

Porting JavaScriptCore to Android

- I led second attempt trying to port existing CMake system
  - I have lots of experience with CMake
  - Despite any ugliness, it gets the job done
- Dependency on libICU
  - Also kind of large and complicated
  - C++
  - Data sets can be ~30MB (Google Play has a 50MB limit)

libICU

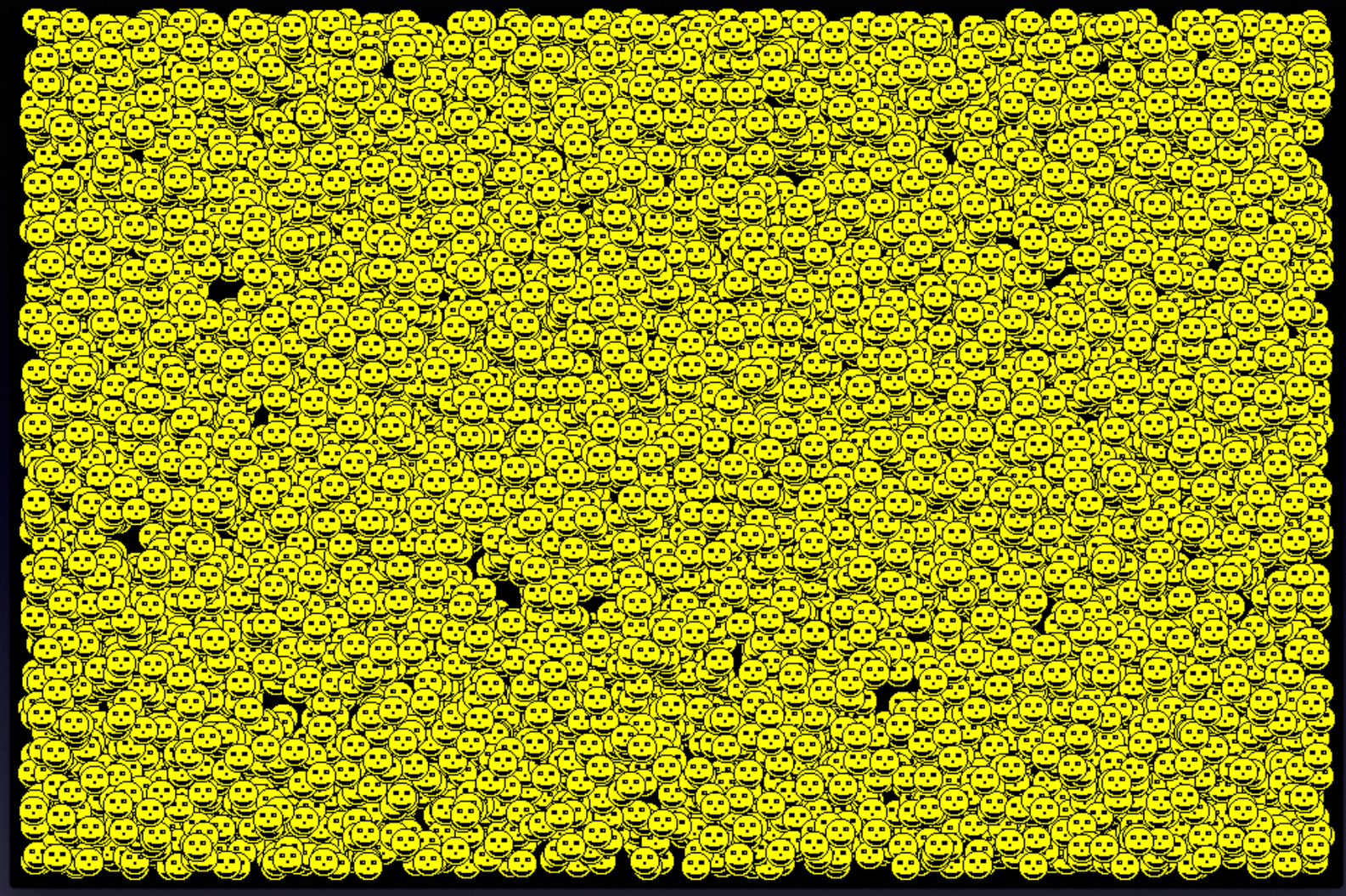
- Binary size due to data set worrisome (dominant factor to size of JSCore)
  - When built as code, size multiples per architecture (armv7, arm64, x86, x86-64)
    - But as standalone data set creates new problems due to the .apk file system (from earlier)
  - Can customize data set to shrink, but don't know enough to what's actually needed
  - Experiments have gotten (JSCore) down to ~5MB safely
    - Think we might be able to get down to ~2MB
- Dynamic library problem:
  - Some Android manufacturers use libICU internally
  - If it is used, when we load our library, the attempt silently fails and we call into the internal one
  - Bad things can happen if the versions don't match
    - (libICU likes to break the API between versions)
  - Ended up static linking

Success

- Got JavaScriptCore built & working on Android in 2 weeks
- Appcelerator team made me feel like a hero
  - Class Act company & people
- But I personally felt guilty, frustrated, angry
  - Lua only took 20 minutes, not 2 weeks
  - Why couldn’t this have been 20 minutes?

WinRT port (Windows Phone)

- Hired outside specialists on WebKit to port
- Took 3-4 months?
- Lots of nasty, hard problems
- Android port benefited from existing Linux port
- But WinRT is a completely different beast and alien even from Win32
- Cost of needing platform specific code
  - (Contrast to Lua's pure ANSI C approach)
- WinRT-ism's like C++CX didn't make things easier



Ultimate Benchmark Showdown

C vs. Lua vs. JavaScriptCore
(SDL bindings)
iOS Android Mac SteamOS/Linux Windows
Max # of sprites at 60 fps

Benchmark Results

	GHz	C	Lua	JavaScript Core	LuaJIT
iPad mini	1.0	750	430	200	430
Nexus 7	1.5	500	300	120	
Fedora 20 GTX560	2.4	42000	3400	2450	
Win 8.1 GTX560	2.4	37000	1950	29	
Mac 10.9 Radeon	3.2	16000	5400	3300	9000
SteamOS HD4000	3.4	5700	2700	3200	3600
Win 8.1 HD4000	3.4	31000	3700	200	18000

Benchmark Results

	GHz	C	Lua	JavaScript Core	LuaJIT
iPad mini	1.0	750	430	200	430
Nexus 7	1.5	500	300	120	
Mac 10.9 Radeon 5670	3.2	16000	5400	3300	9000
SteamOS HD4000	3.4	5700	2700	3200	3600
Win 8.1 HD4000	3.4	31000	3700	200	18000

Benchmark Takeaways

- C > LuaJIT > JavaScriptCore (JIT) > Lua > JavaScriptCore (no-JIT)
- Slower systems: Lua is ~ 60% of C, JS is 50% of Lua
- Faster systems: GPU/drivers is bottleneck in C, Lua/JS causing CPU bottlenecks
- Android results fishy…
  - Could be optimization flags
  - Could be higher screen resolution (GPU bottleneck)
- Windows JavaScriptCore bad performance was a bug that was since fixed
- Windows Lua suspect: Might be debug build or something is wrong

Benchmark Takeaways

- JavaScriptCore JIT appears disabled on Mac
  - Linux beats Mac: Apple should be embarrassed
- JavaScriptCore JIT faster than Lua, but not by a lot
- LuaJIT JIT is impressive when enabled, but same as Lua when not
  - LuaJIT still did not match C
    - Note: Using standard C bindings, not ffi.* bridge

Benchmark Takeaways

- Remember that LuaJIT cannot optimize C calls or anything in C
- Despite being a straight binding, there is still a lot of code even in this simple program that LuaJIT optimized

Benchmark Takeaways

- Valve confirmed to me that the Intel HD4000 drivers are very good
- There is a strange Windows performance problem which looks like Power Management is slowing down the CPUs

Benchmark Conclusions

- Numbers still closer/more reasonable to expected than before
    - Original prototype from beginning could do only 80 sprites at 30 fps (60 fps did not work at all)
    - Now 200 at 60 fps in JS on iPad mini (slowest device)
- Overall, this was a rare side-by-side comparison of how much overhead scripting creates

What does this mean?

- Despite the performance spread, there is enough horsepower to do real games with a buttery smooth 60 fps in JavaScriptCore (and without JIT)
  - Just as we know we can do with Lua
  - Even on iOS & Android
  - Even on a Raspberry Pi
- Don’t be deceived. Despite larger heft of JS VM (compared to Lua), it is a tiny fraction of the web browser
  - Don’t have the memory pressure bloat from the browser
  - Don’t have multiple tabs & windows to manage
  - Garbage collector under less pressure with more free RAM

Conclusion

- I'd probably be much happier if we used Duktape
- Lua is really nice and I missed it
  - Didn't get to talk about API differences, but I like Lua's stack approach
- Writing JavaScript code from the C-side is harder
  - Not clear you can always do something on the native side of the bridge
  - Lua's stack based C API can do everything
  - Aside, never figured out how to override '===' in JSCore
    - (If you know, please contact me)
  - Also lack of weak tables in JS (at the time) is painful

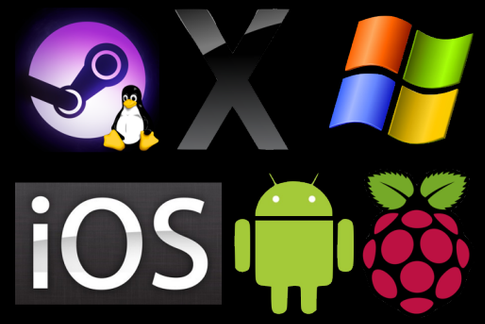
Lua design appreciation

- Lua is pure C (no lib C++ hell)
- Lua has minimal dependencies (nothing like libICU)
    - Even its use of the C standard library is restrained (minimal Bionic risks)
- Lua is optimized for embedding (fastest calls across the bridge)
- Lua has a nice C API that is considered part of the language
    - Good documentation on how to use too
- Lua is easy to build (20 minutes vs. 2 weeks or 3 months)
- Simple, small, and focused are undervalued virtues
    - Less is more

Frustrations

- Software is too thick/bloated, too much abstraction
  - Can’t understand it & can’t modify it & integration too hard
  - Performance is a lost art & misunderstood by most (e.g. understanding the hardware matters)
  - Want cross-platform desktop AND mobile
    - (didn’t have time to discuss why mobile-only was a problem)
- Tired of the religious Language Wars

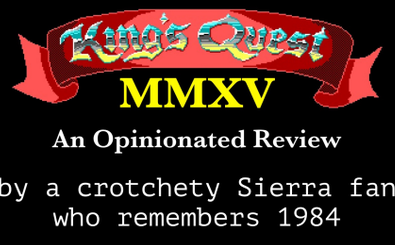
Blurrr SDK



- C Lua JavaScript Swift
- iOS Android macOS SteamOS/Linux Windows Raspberry Pi
- I'll write the high performance engine core in pure C using Data Oriented Design principles and SIMD
- You pick the language you want to use it with
- 2D games, SDL ecosystem
  - IUP (previous talk) possible direction to do regular apps too

Links

- Eric Wing (@ewingfighter)
  - Website: http://playcontrol.net
  - YouTube: https://www.youtube.com/user/ewmailing



  - Now playing: "***Why we loved Sierra Games***"

- Blurrr SDK (my current project)
  - https://BlurrrSDK.com, @BlurrrSDK

- GIST Cancer Research Fund
  - http://www.gistinfo.org

Carlos M. Icaza

Adobe

Macromedia

Corona SDK

Platino

@codinginswift