Accelerating Android Builds

Eric Melski, Chief Architect
Electric Cloud
ericm@electric-cloud.com

About Electric Cloud

- **Leaders in Software Delivery Acceleration and Automation**
  - Helping large-scale, Fortune 500 achieve demonstrable results
  - 9 patents in the domain of parallel computing, build acceleration

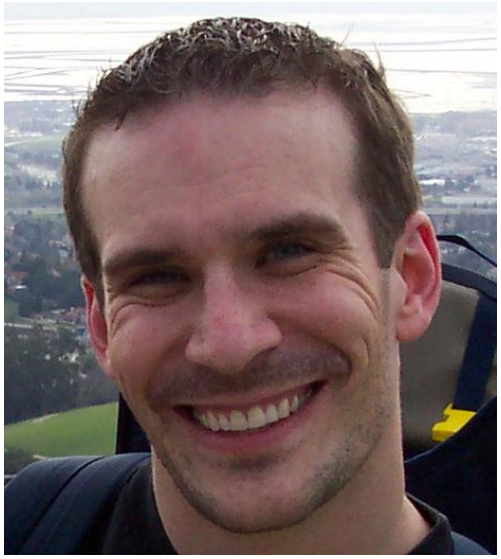
- Created **ElectricAccelerator** in 2002
  - Ground-up reimplementation of GNU make
  - Faster builds via parallel and distributed processing – with a twist!
  - **Dependency detection and correction means builds never break**[1]
  - Used by hundreds of companies, thousands of users, millions of builds!

[1] Due to execution ordering problems

electric CLOUD

Who is Eric Melski?

- Chief Architect at Electric Cloud
  - Responsible for ElectricAccelerator and ElectricInsight
  - Founding member of Electric Cloud in 2002
  - More than 10 years experience analyzing and accelerating builds



@emelski

http://blog.melski.net

[1] Source: Gartner, February 2013

electric CLOUD

Why look at Android builds?

- Android is everywhere
  - 68% of global mobile phone market share[1]
  - Explosion of non-phone applications
    - Automotive (Saab, Renault)
    - Entertainment (Ouya, Nvidia)
    - Exercise equipment (NordicTrack)
    - Rice cookers! (Panasonic)
- **Android is everywhere**
  - … and that means **thousands of developers building Android**
- **What if we can make those builds faster?**
  - How would *your* development process change?
  - How much more could you achieve?

[1] Source: Gartner, February 2013

electric CLOUD

What is (and is not) covered in this talk?

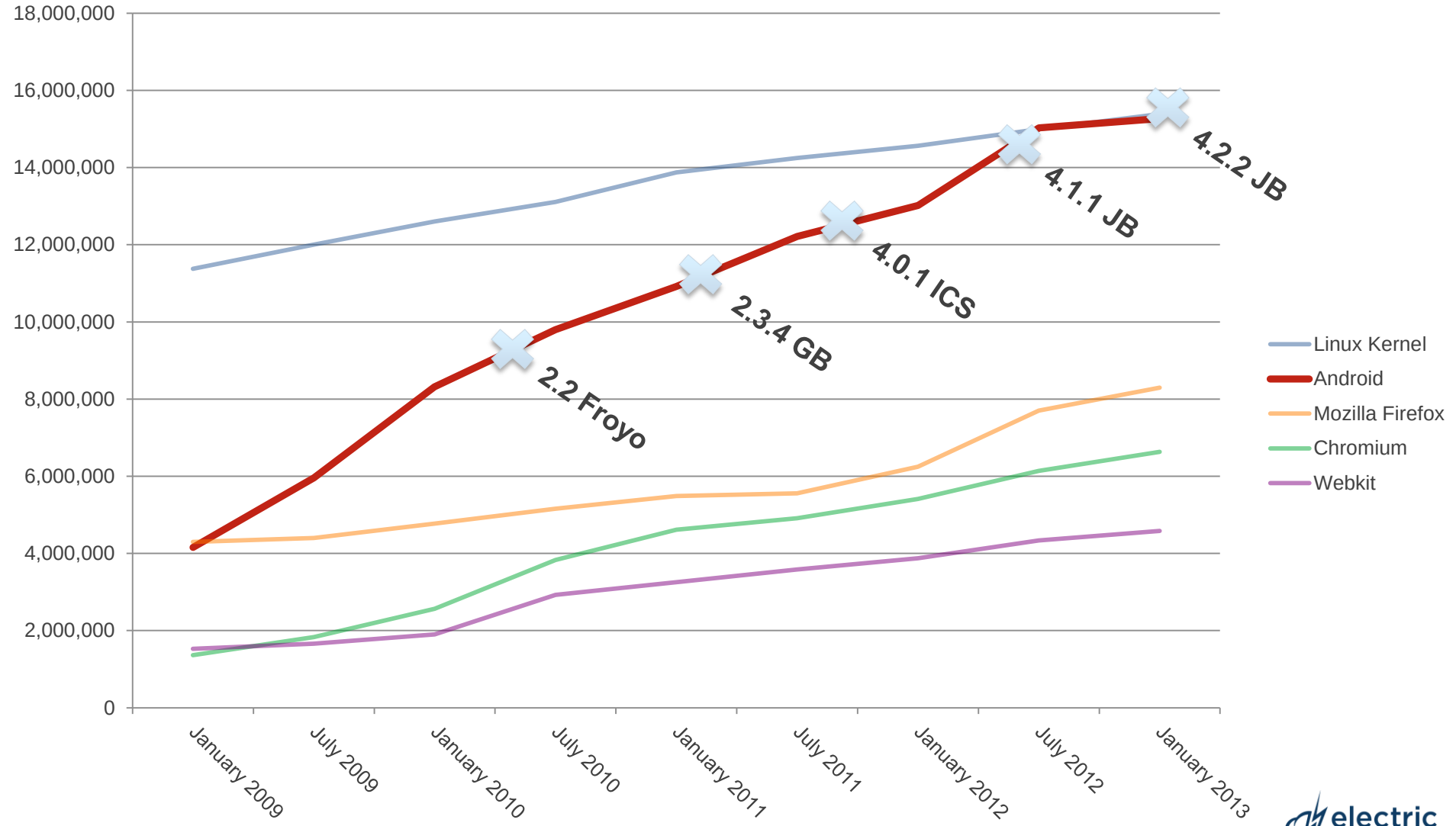
- Build acceleration techniques fall into two broad categories:
  - **Hardware** improvements like faster CPU's and better disks
  - **Software** improvements like smarter build tools and faster compilers
- In general, hardware and software improvements are complimentary
  - If you want the fastest possible builds, leverage *both!*

- This talk is about **software** techniques for build acceleration
  - Given a *fixed hardware platform*, how fast can we build Android?
  - Or, **what do you do when you reach the limits of your hardware?**

[1] Source: Gartner, February 2013

AOSP LOC Evolution

Jan 2009 – Jan 2013[1]

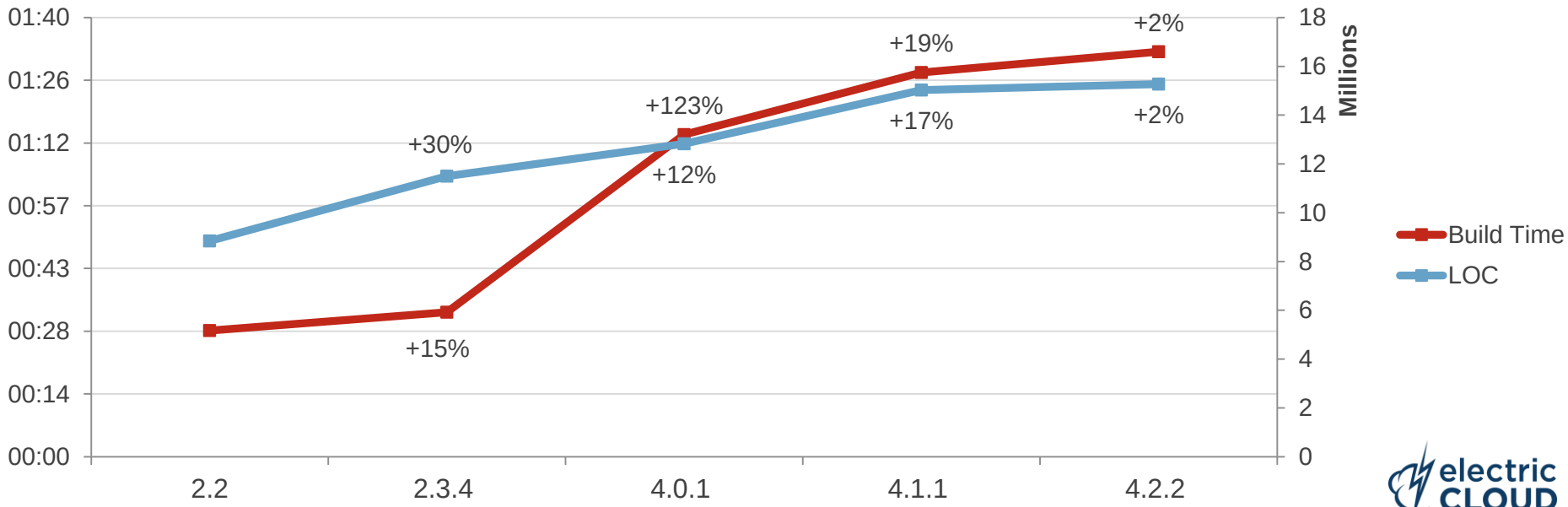


[1] Source: http://www.ohloh.net, http://en.wikipedia.org/wiki/Android_version_history (February 2013)

electric CLOUD

AOSP Build Evolution

Android Version	Release Date	LOC[1]	LOC Growth %	GNU Make Build Time[2]	Build Time Growth %
2.2	May 2010	8,837,858	-	28m55s	-
2.3.4	April 2011	11,492,324	30%	33m10s	15%
4.0.1	October 2011	12,827,330	12%	1h13m54s	123%
4.1.1	July 2012	15,028,331	17%	1h28m11s	19%
4.2.2	February 2013	15,266,803[3]	2%	1h32m56s	2%

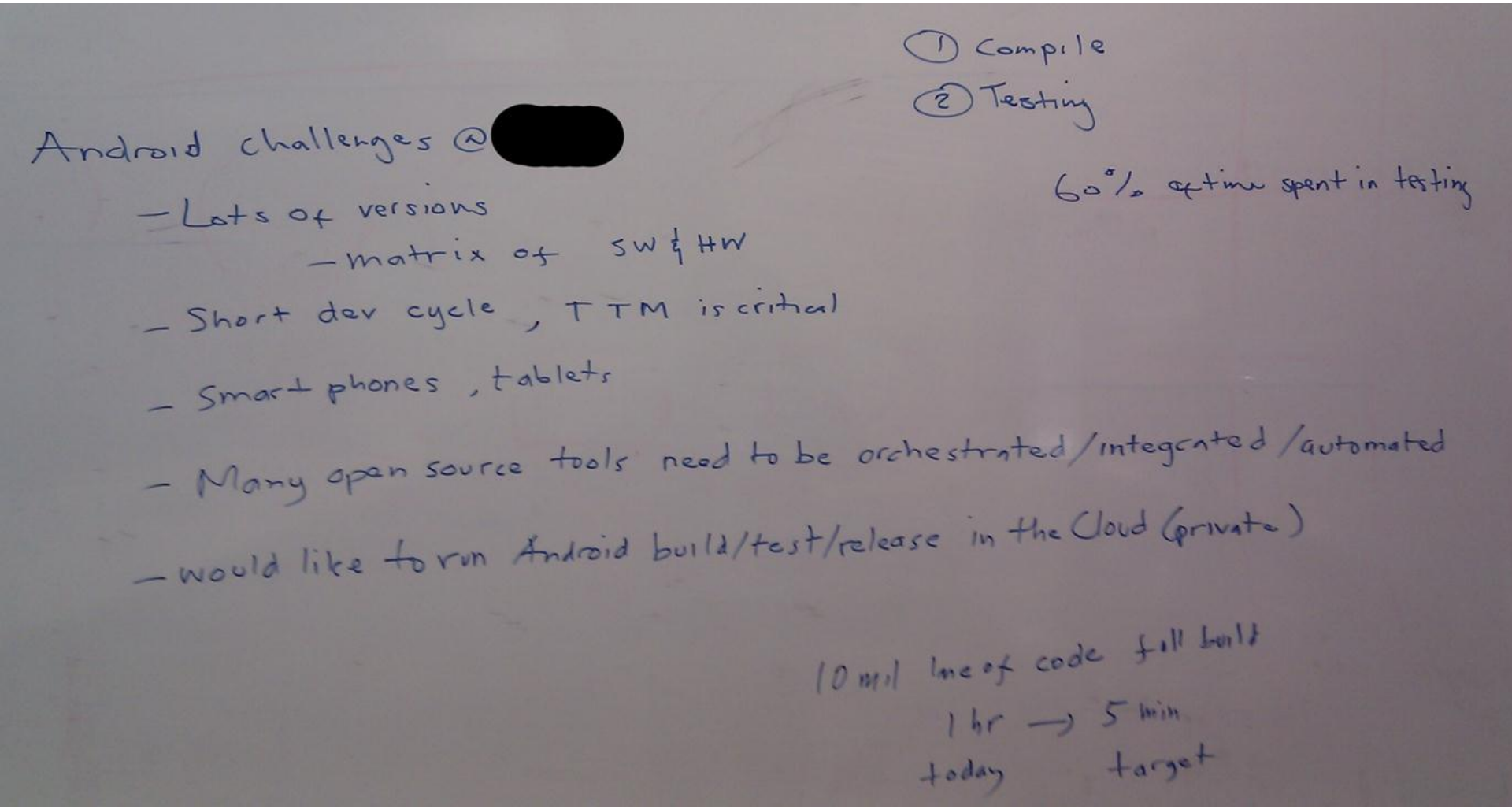


[1] http://www.ohloh.net, [2] Builds performed on 8-core 16GB RAM server, [3] LOC data as of December 2012

electric CLOUD

Android Development Landscape

Real-world challenges at large APAC mobile device maker



electric CLOUD

Common wisdom about Android builds

- I can just…
  - … add more cores
  - … use distcc
  - … use ccache
- “The Android build is as optimized as it can be!”

electric CLOUD

The common wisdom is wrong.

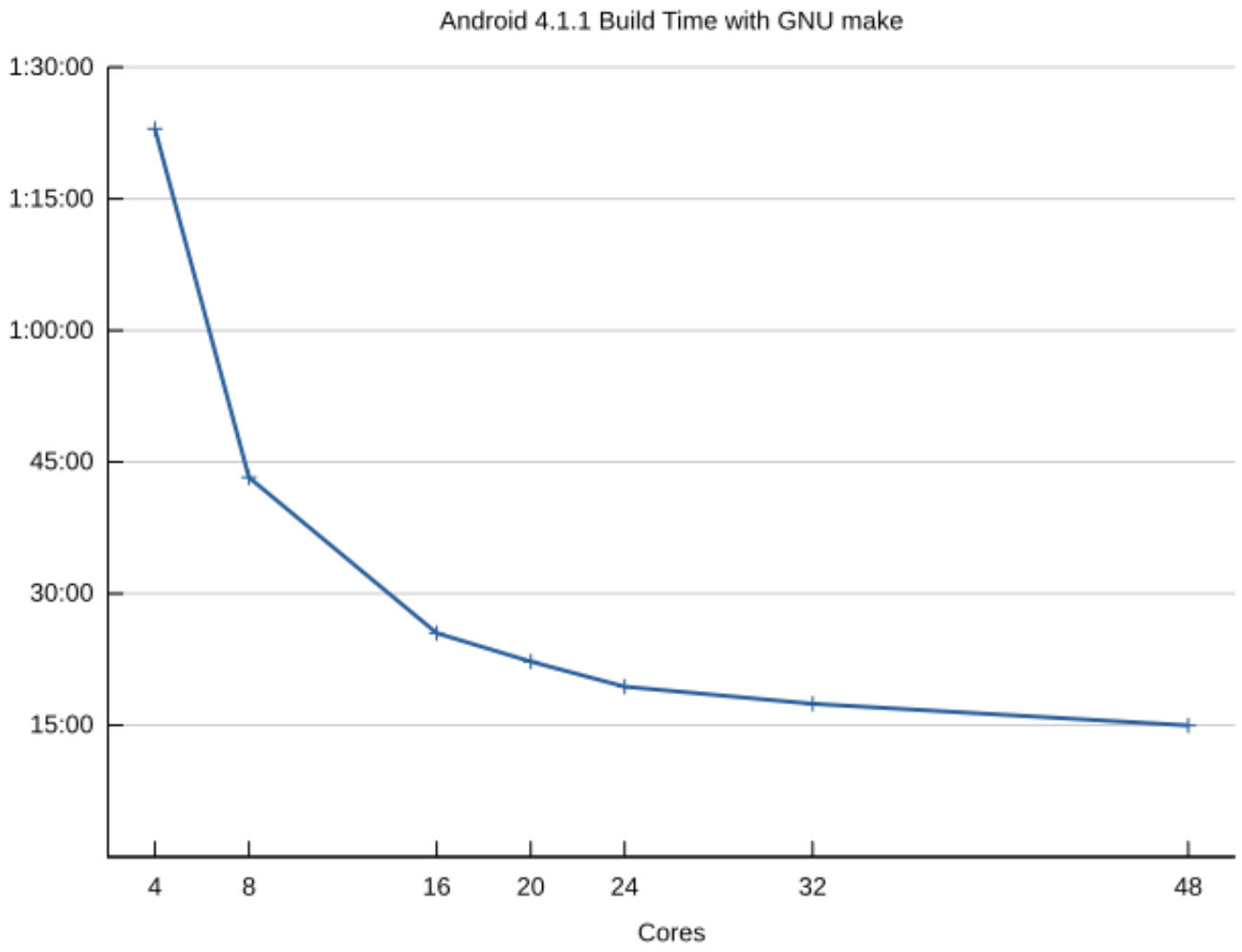
Sidebar: how much faster is “interesting”?

- Some large mobile device makers run > 50K builds/week
  - At 15 minutes per build that’s **12,500 hours of build time**
  - A reduction of just *one minute* would save **800 hours every week**
- What about wait time?

- Faster builds = more iterations
- Faster builds = higher quality
- Faster builds = lower time to market

electric CLOUD

How fast is GNU make?

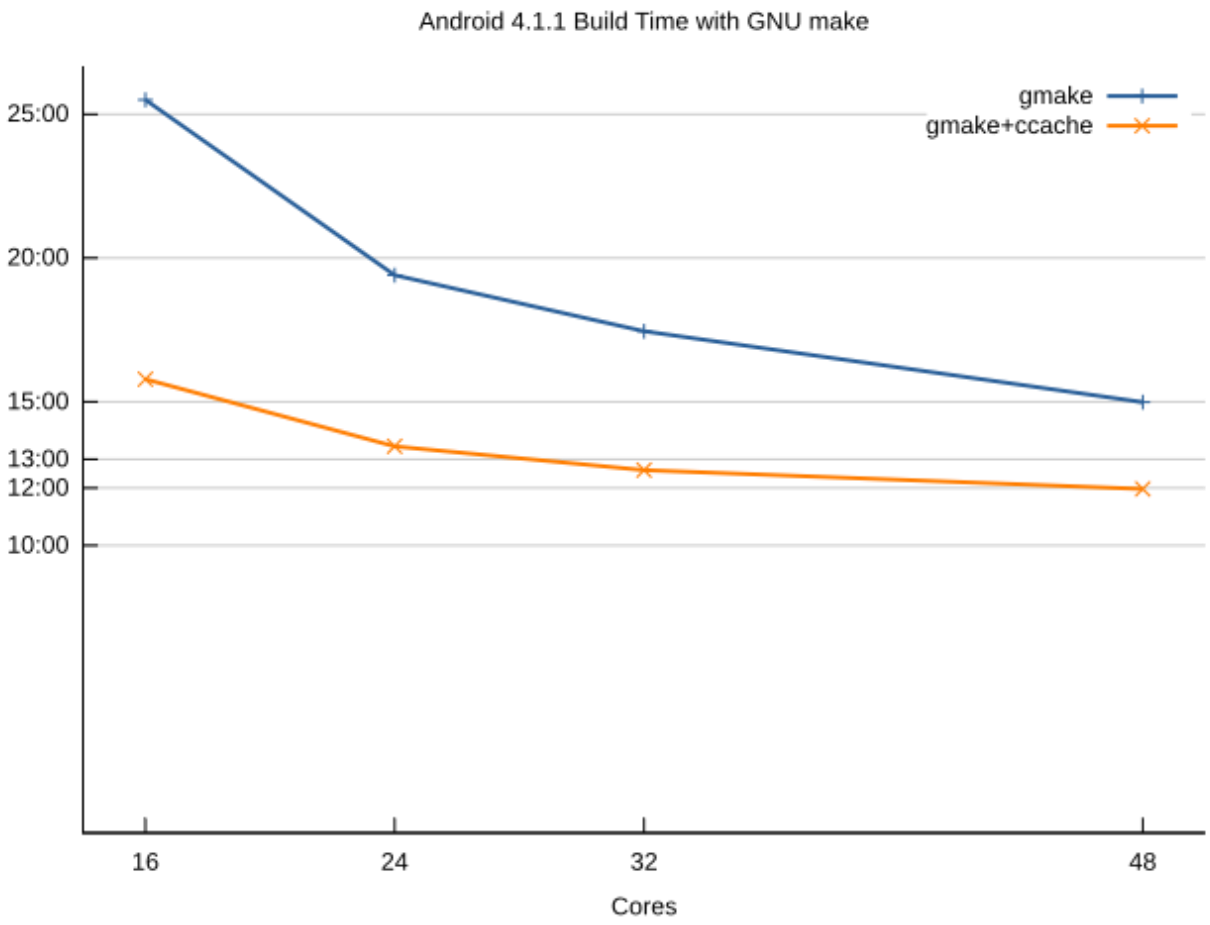


- **Best time: ~15m**
- 48-cores
- 128GB RAM
- No ccache

electric CLOUD

What if we add ccache?

- ccache helps, but not as much as you might hope



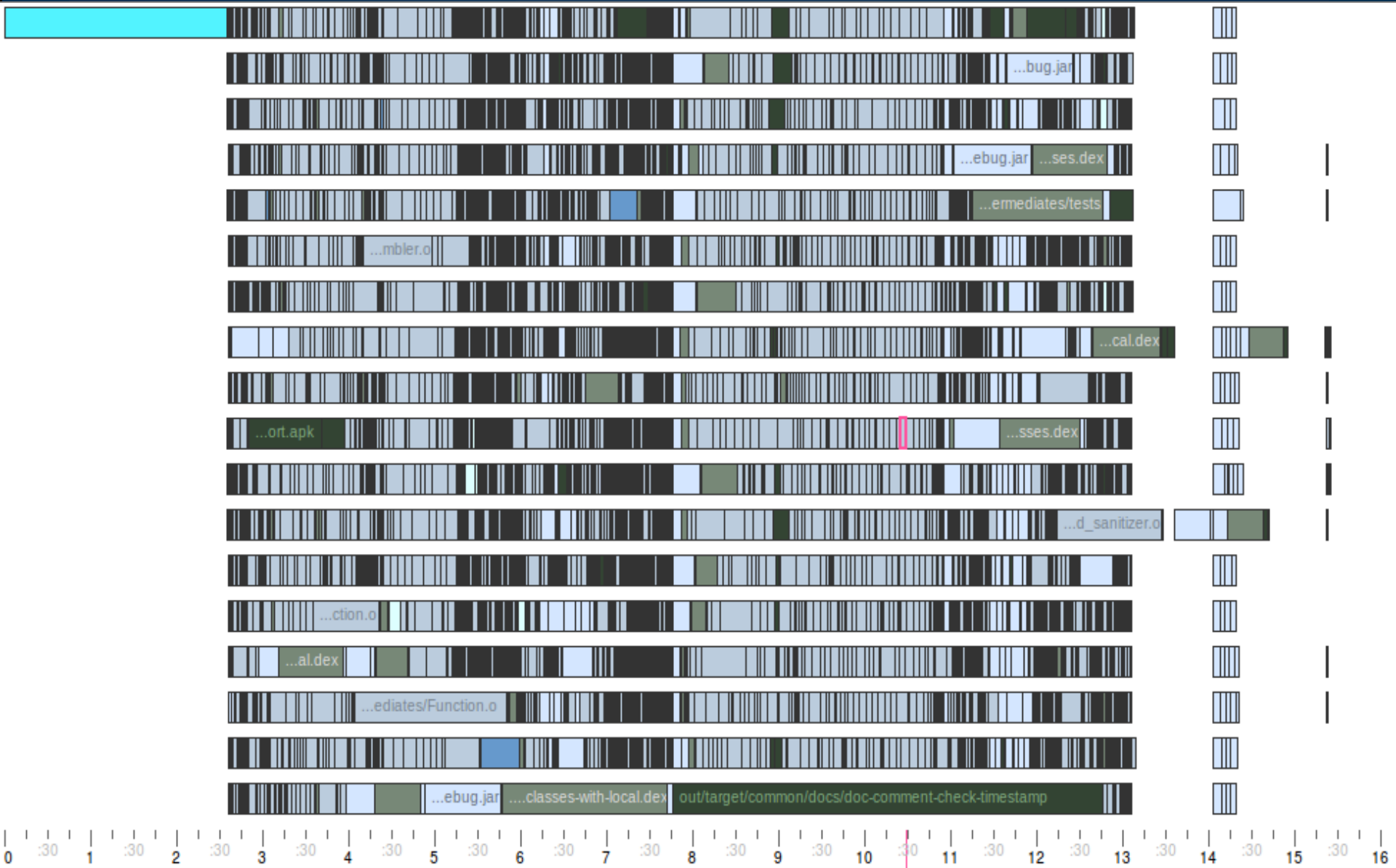
electric CLOUD

Problem: Visibility

- Knowing the end-to-end time is great but not *actionable*
- **We need visibility**
  - Where is the time going?
  - What are the bottlenecks?
  - Are there opportunities for improvement?
- How do we get that visibility?
  - Instrument the build… *somehow!* strace, echo, hack gmake, or…

- **ElectricInsight**
  - Visualization and analysis of builds run with ElectricMake
  - **Let's pump this build through emake and see how it looks!**

Solution: ElectricInsight



What can we see from here?

- An ideal build would look like a tightly packed box
- Overall looks almost perfect – well done, Android team!

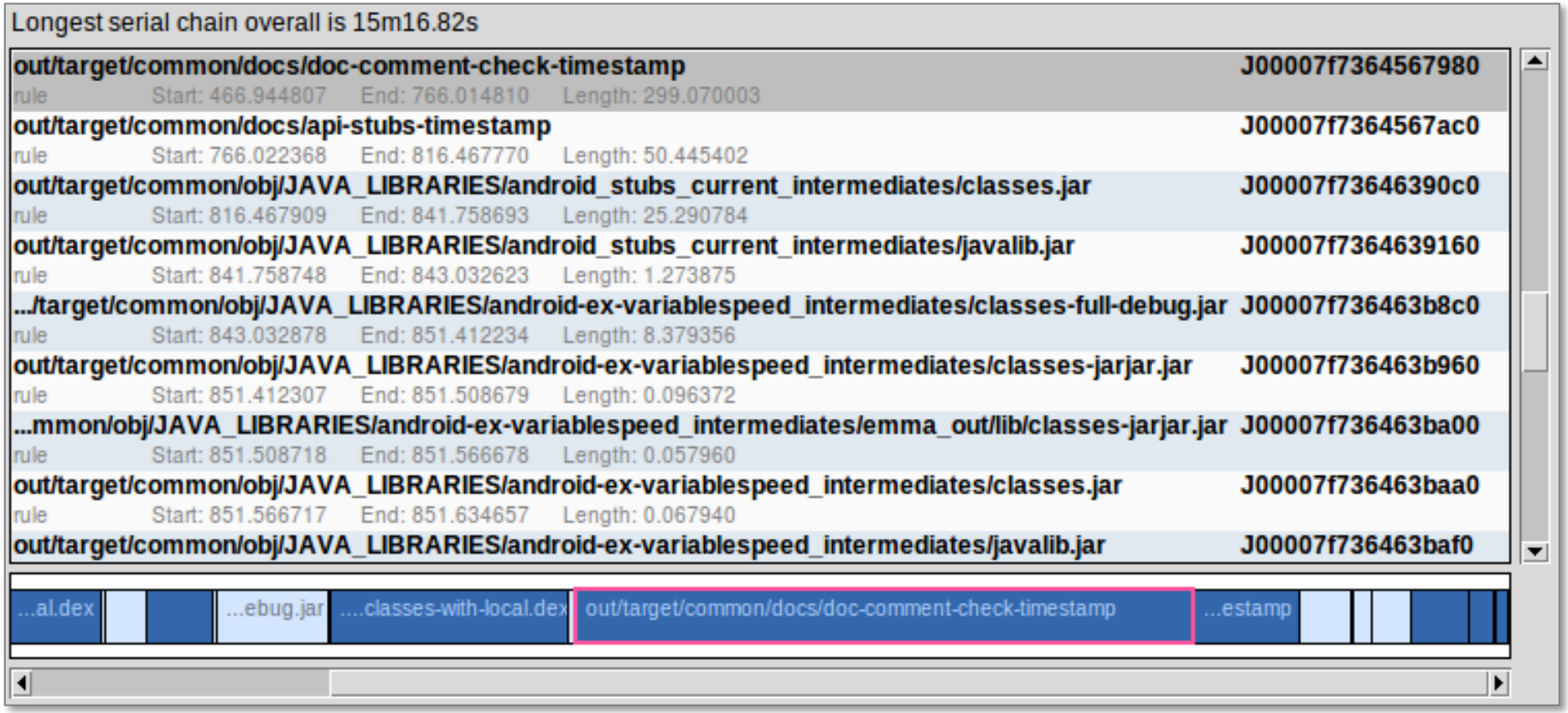
- **But!** a few things stand out:
  - Crazy long parse time[1]
  - Gaps at the end of the build, indicative of serializations
  - Some very long jobs, like `doc-comment-check-timestamp`

- We'll look at each, but first: what if we just use more cores?

[1] emake parsing may be slower than gmake parsing

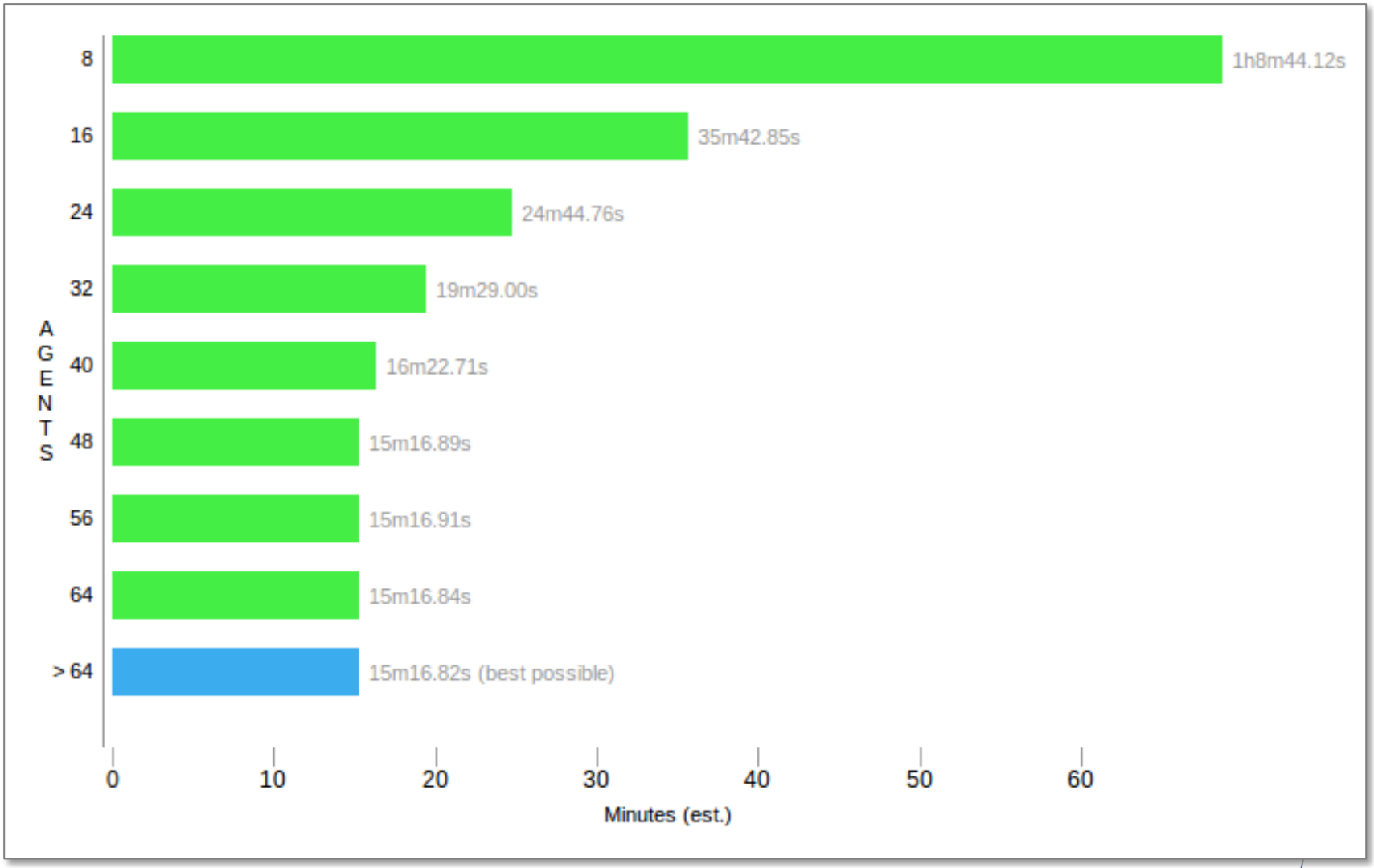
Longest serial chain

Best possible is about 15m



electric CLOUD

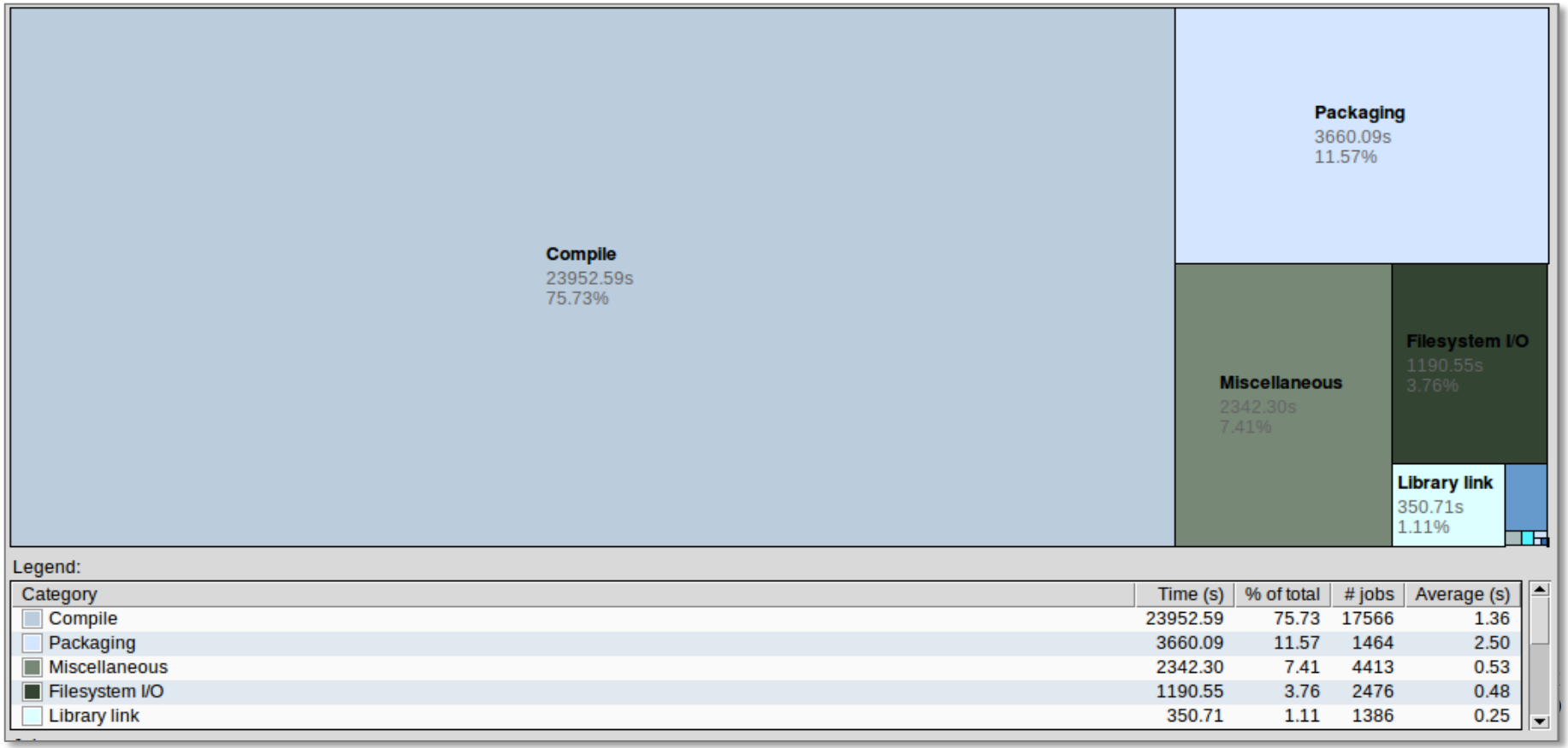
Projected runtime with more cores



electric CLOUD

Why doesn't ccache help more?

- Lots of non-compile work in the build
- Most compiles are already pretty fast (1.3s average)
- *Under ideal conditions for ccache,* best improvement is about 4x!
  - Serial build
  - Completely full cache (make ; make clean ; make)



Legend:

Category	Time (s)	% of total	# jobs	Average (s)
Compile	23952.59	75.73	17566	1.36
Packaging	3660.09	11.57	1464	2.50
Miscellaneous	2342.30	7.41	4413	0.53
Filesystem I/O	1190.55	3.76	2476	0.48
Library link	350.71	1.11	1386	0.25

Problem: long parse time

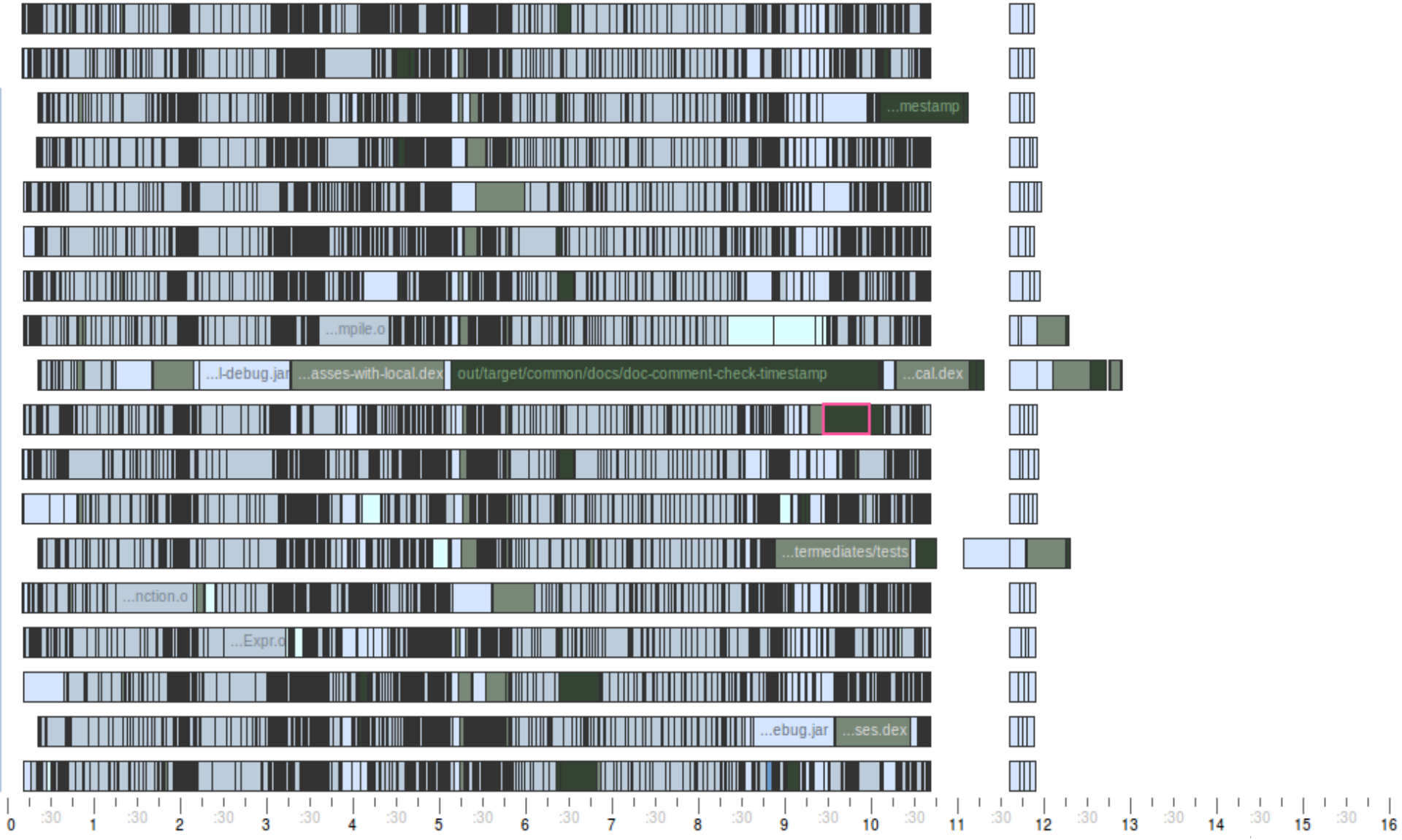
- Why do we build?
  - To transform sources into deliverables (programs, docs, etc).
- Does parsing makefiles transform sources into deliverables?
  - **Nope.**
- **Parsing makefiles is pure overhead**
  - But you have to tell make what to do *somehow*
  - Unless you want to manually invoke the compiler, linker, etc.

Solution: parse avoidance

- What if we don't parse (every time)?
  - Makefiles don't change very often, so why reparse every time?
- Reuse parse results from a previous build, as long as…
  - Makefiles are unchanged (MD5)
  - Command-line is unchanged
  - Environment is unchanged
- How do we do it?
  - Electric Make already has parse results in a reloadable form, just need to add cache management
  - GNU make doesn't, but could be modified

Parse avoidance impact

Build time reduced to about 13m30s

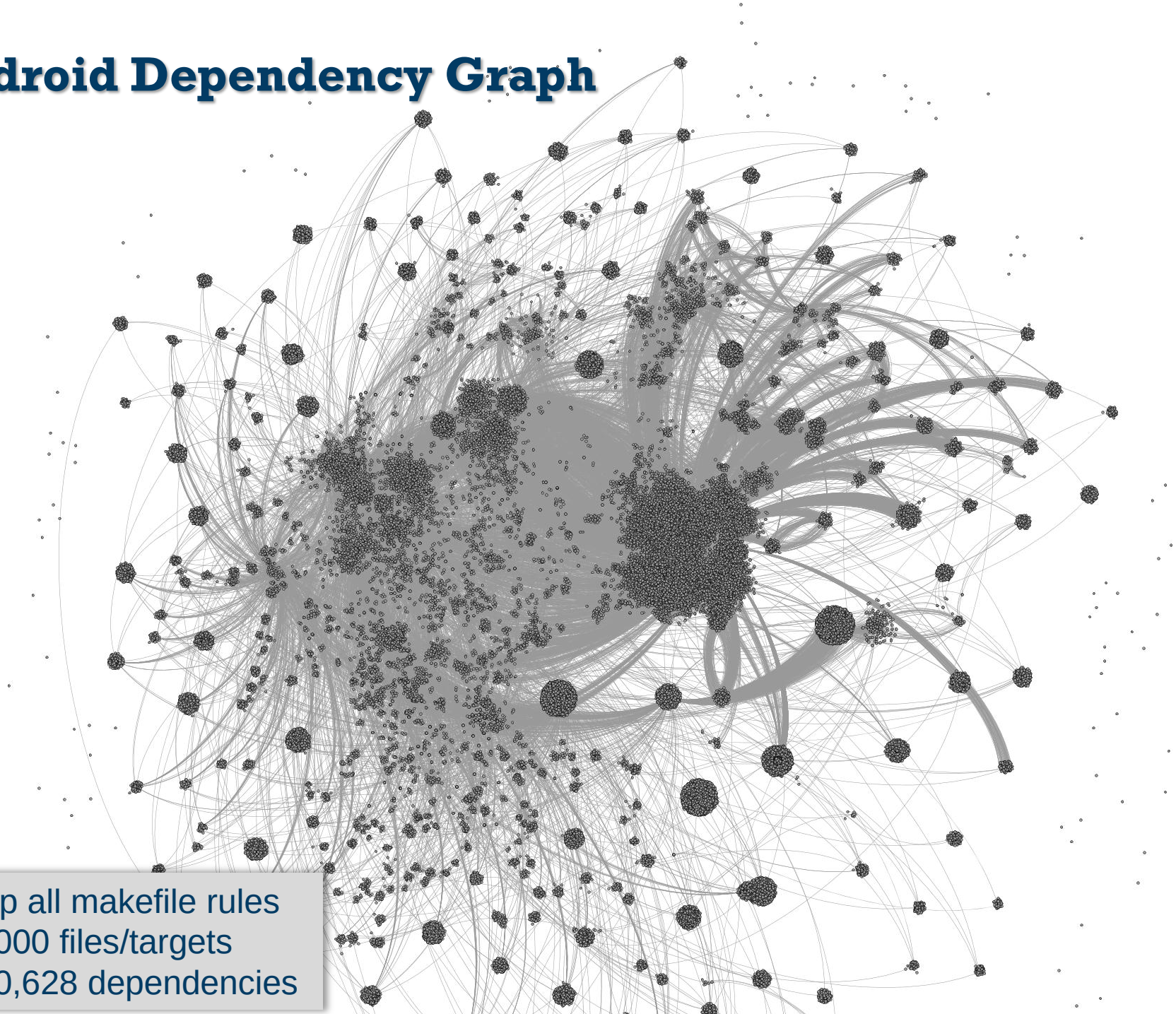


Problem: serializations

- Gaps in visualization suggest serializations
- Q: How many dependencies are there in the Android build?
- A: **More than you think!**

electric CLOUD

Android Dependency Graph



- Dump all makefile rules
- 100,000 files/targets
- 1,990,628 dependencies

Dependencies in Android

- ~19 dependencies per file: why so many?
- Consider a typical build structure:

```
lib.a: foo.o bar.o
foo.o: foo.c foo.h util.h
bar.o: bar.c bar.h util.h
```

- Some files will have many dependencies
- Most have only a few
- What is going on in Android?

Superfluous dependencies in Android

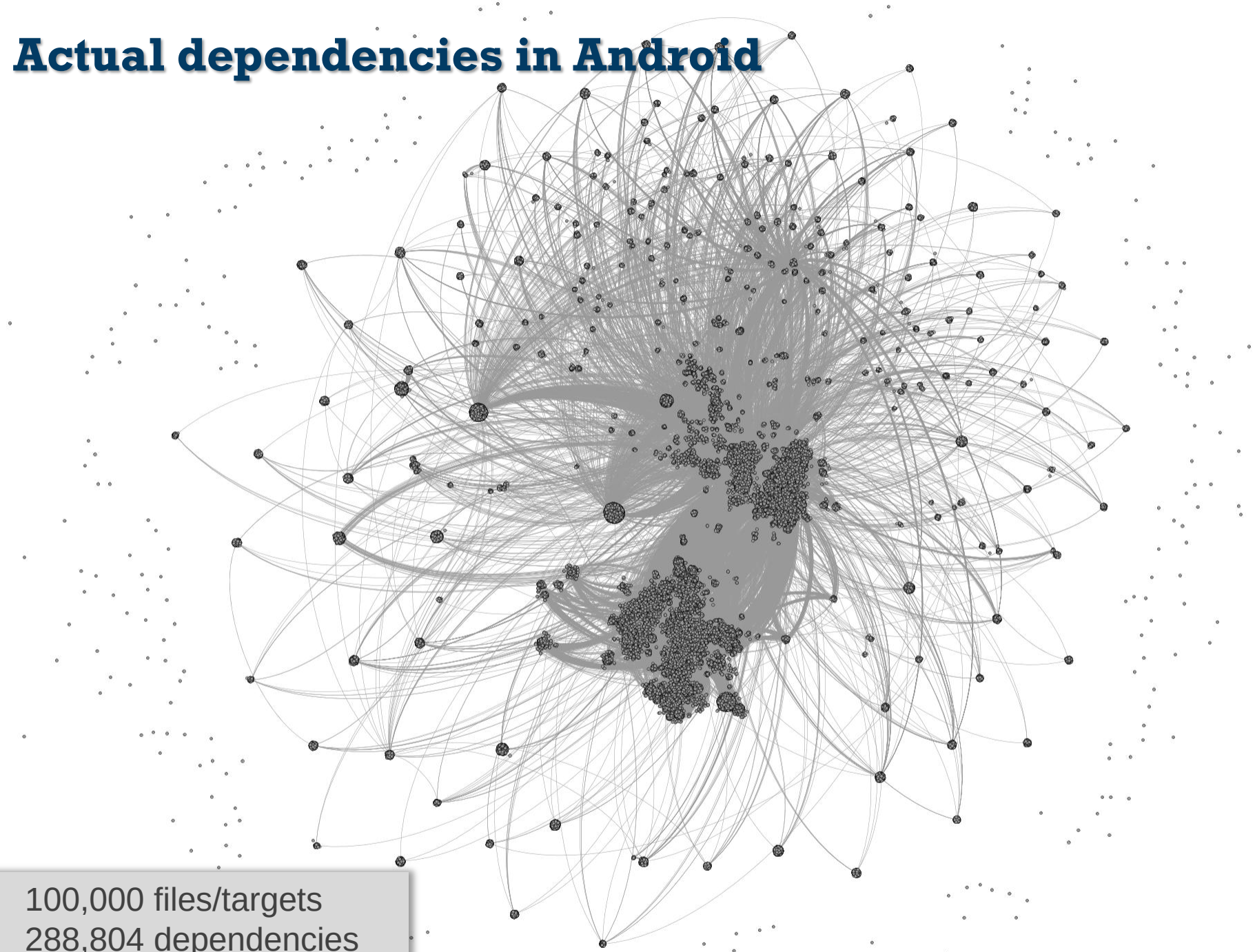
- Do we really need all 1.9M dependencies?
- The filesystem can tell us!
  - Collect a list of files *actually* used to generate a target
  - Compare to the list of prerequisites specified in the makefile
- Example:

```
foo.txt:
    echo “Hello” > foo.txt

bar.txt: foo.txt
    echo “World” > bar.txt
```

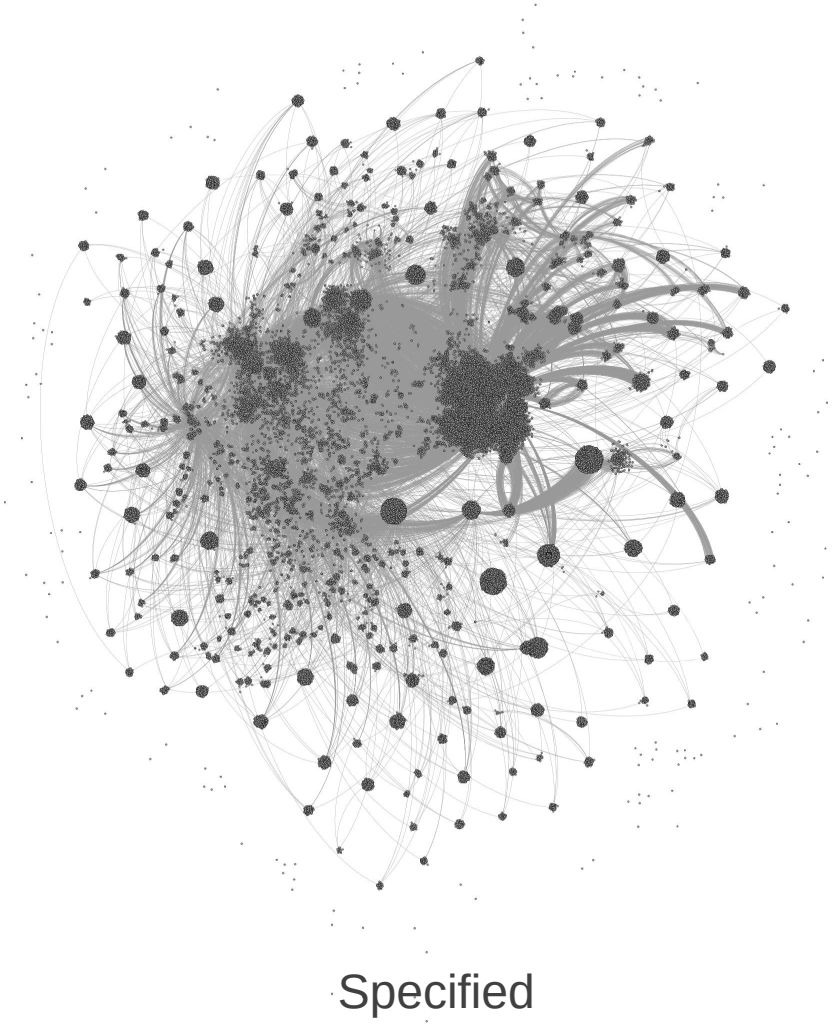
- Why not run foo.txt and bar.txt in parallel?

Actual dependencies in Android

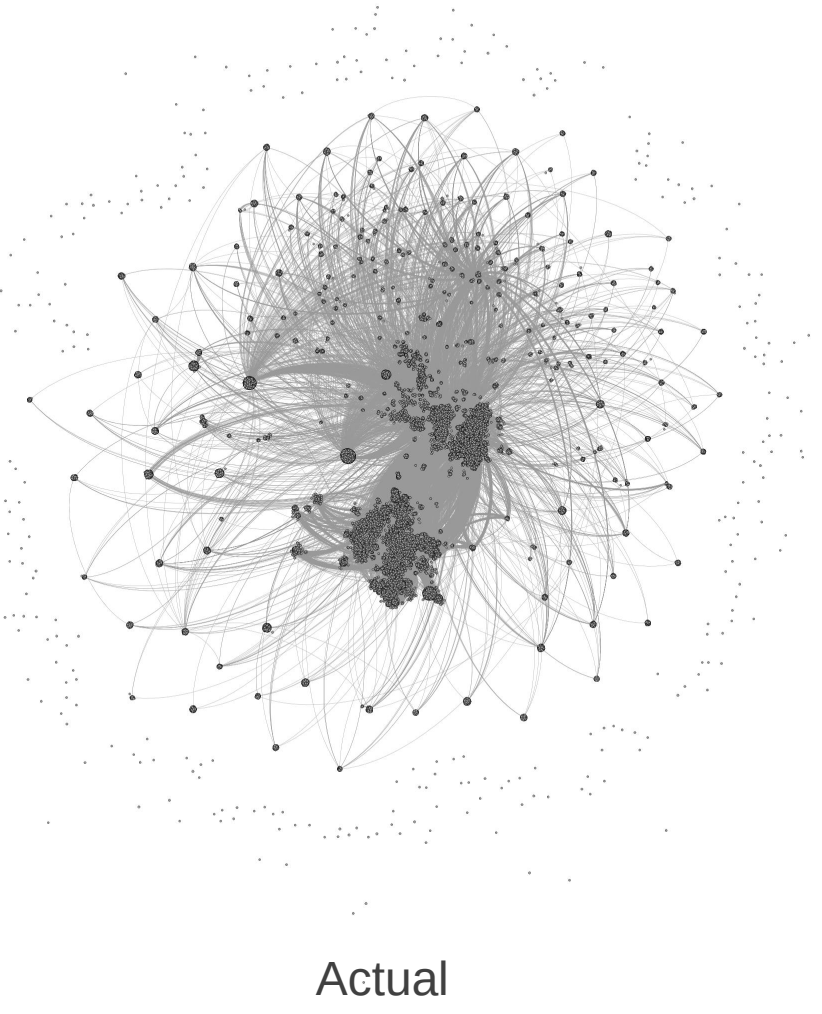


- 100,000 files/targets
- 288,804 dependencies

Specified vs. Actual dependencies in Android



Specified



Actual

Solution: dependency optimization

- Impossible to manually eliminate superfluous dependencies
- Electric Make can prune them automatically
- If you use gmake, you can achieve the same effect:
  - Remove existing dependency specifications
  - Generate minimal dependency specifications from emake data
  - Before:

```
$(OBJECTS): %.o: %.cpp $(ALL_HEADERS) $(OTHER_DEPS)
    $(COMPILE.CC)
```

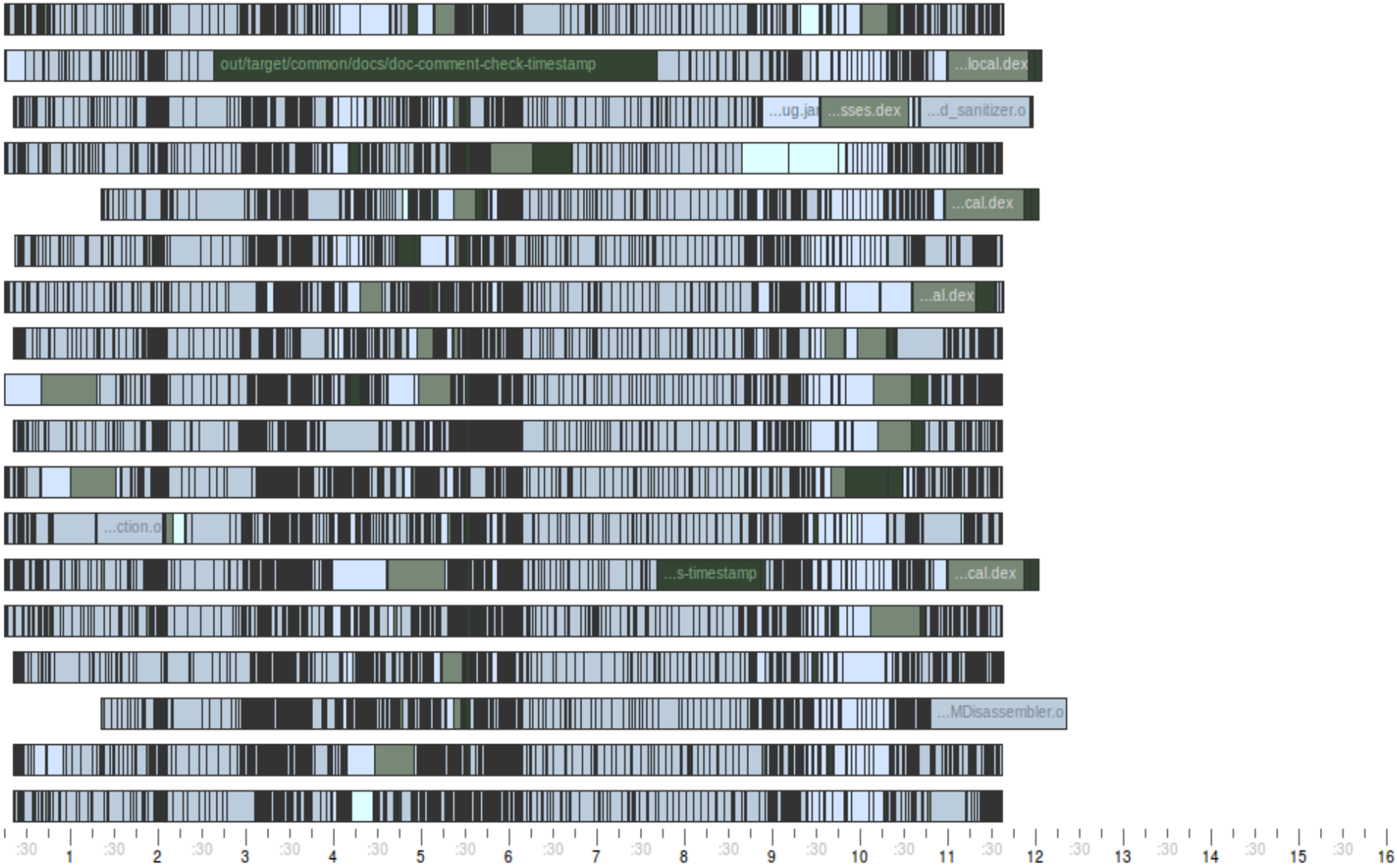
  - After:

```
$(OBJECTS): %.o: %.cpp
    $(COMPILE.CC)
foo.o: foo.h util.h
bar.o: bar.h util.h generated.h
```

electric CLOUD

Dependency optimization impact

Build time reduced to about 12m30s



Problem: long jobs

- Several long jobs in Android build:

Longest jobs:

out/target/common/docs/doc-comment-check-timestamp				**J00007f7364567980**
rule	Start: 466.944807	End: 766.014810	Length: 299.070003	
<no name>				**J0000000002188a20**
parse	Start: 0.512912	End: 155.146917	Length: 154.634005	
out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/noproguard.classes-with-local.dex				**J00007f73645677a0**
rule	Start: 347.303202	End: 462.336772	Length: 115.033570	
out/target/product/generic/obj/STATIC_LIBRARIES/libLLVMCore_intermediates/Function.o				**J00007f7364596fa0**
rule	Start: 244.301372	End: 350.684783	Length: 106.383411	
out/host/linux-x86/obj/EXECUTABLES/vm-tests-tf_intermediates/tests				**J00007f736476ed50**
rule	Start: 675.592890	End: 766.461396	Length: 90.868506	
...et/product/generic/obj/STATIC_LIBRARIES/libLLVMARMDisassembler_intermediates/ARMDisassembler.o				**J00007f73647c6500**
rule	Start: 743.108224	End: 829.887677	Length: 86.779453	
out/target/product/generic/obj/SHARED_LIBRARIES/tsan-arm-linux_intermediates/thread_sanitizer.o				**J00007f73647c55b0**
rule	Start: 734.370536	End: 807.504772	Length: 73.134236	
out/host/common/obj/JAVA_LIBRARIES/apache-harmony-tests-hostdex_intermediates/classes.dex				**J00007f7364752510**
rule	Start: 694.791112	End: 750.286637	Length: 55.495525	
out/host/common/obj/JAVA_LIBRARIES/core-tests-hostdex_intermediates/classes-full-debug.jar				**J00007f7364752bf0**
rule	Start: 662.566889	End: 716.326949	Length: 53.760060	
out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/classes-full-debug.jar				**J00007f7364567480**
rule	Start: 293.147110	End: 346.482869	Length: 53.335759	

- Long jobs impose a lower bound on speed
- `doc-comment-check-timestamp` is nearly 5m!

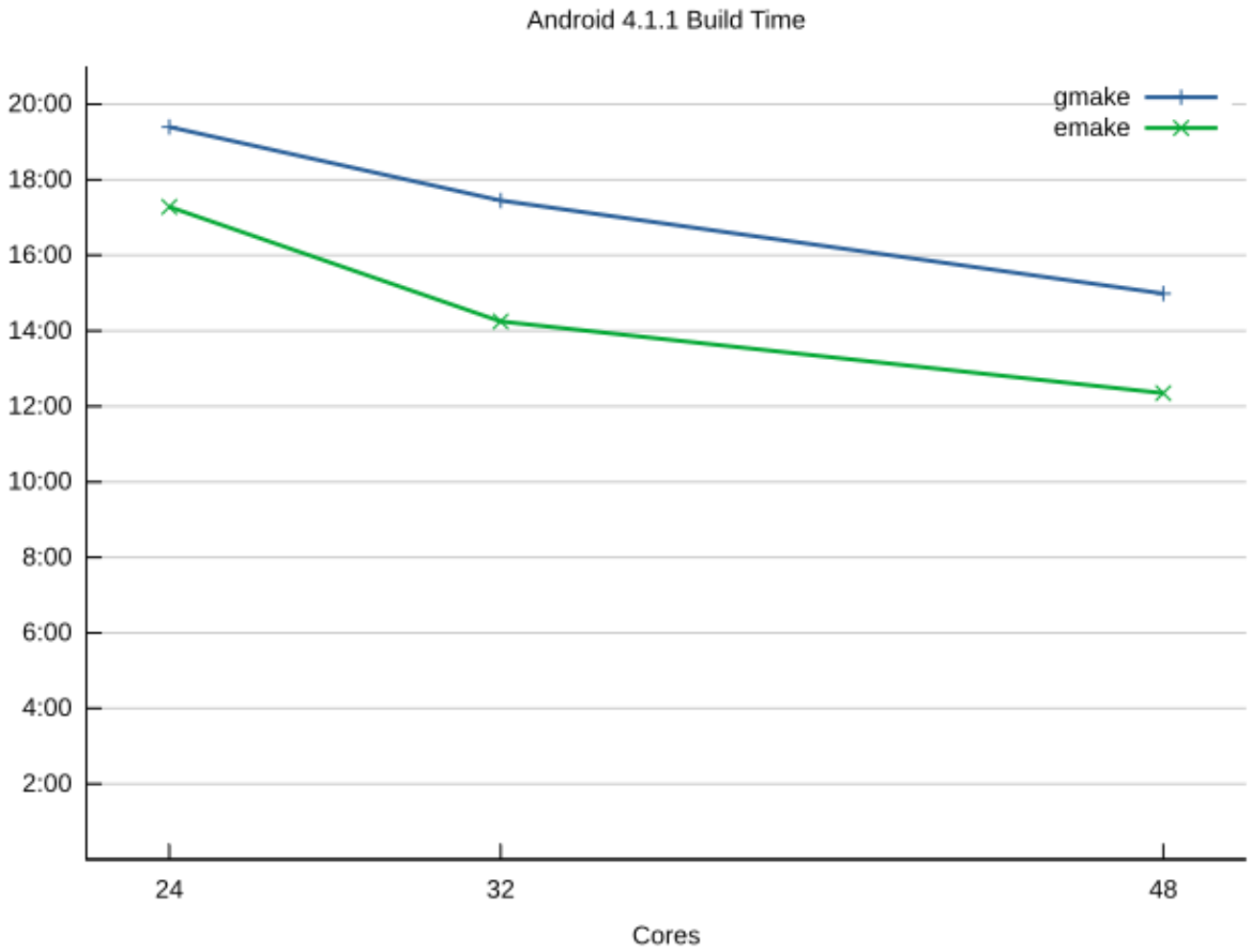
electric CLOUD

Solution: cache javadoc (TBD)

- Concept: like ccache, but for javadoc
- Cache output, reuse in next build if…
  - Command-line args match
  - Input file MD5's match
- Conservatively estimate this could cut 45-60s
  - Javadoc job itself could be reduced by 4m or more
  - End-to-end impact is less due to parallelism

Summary

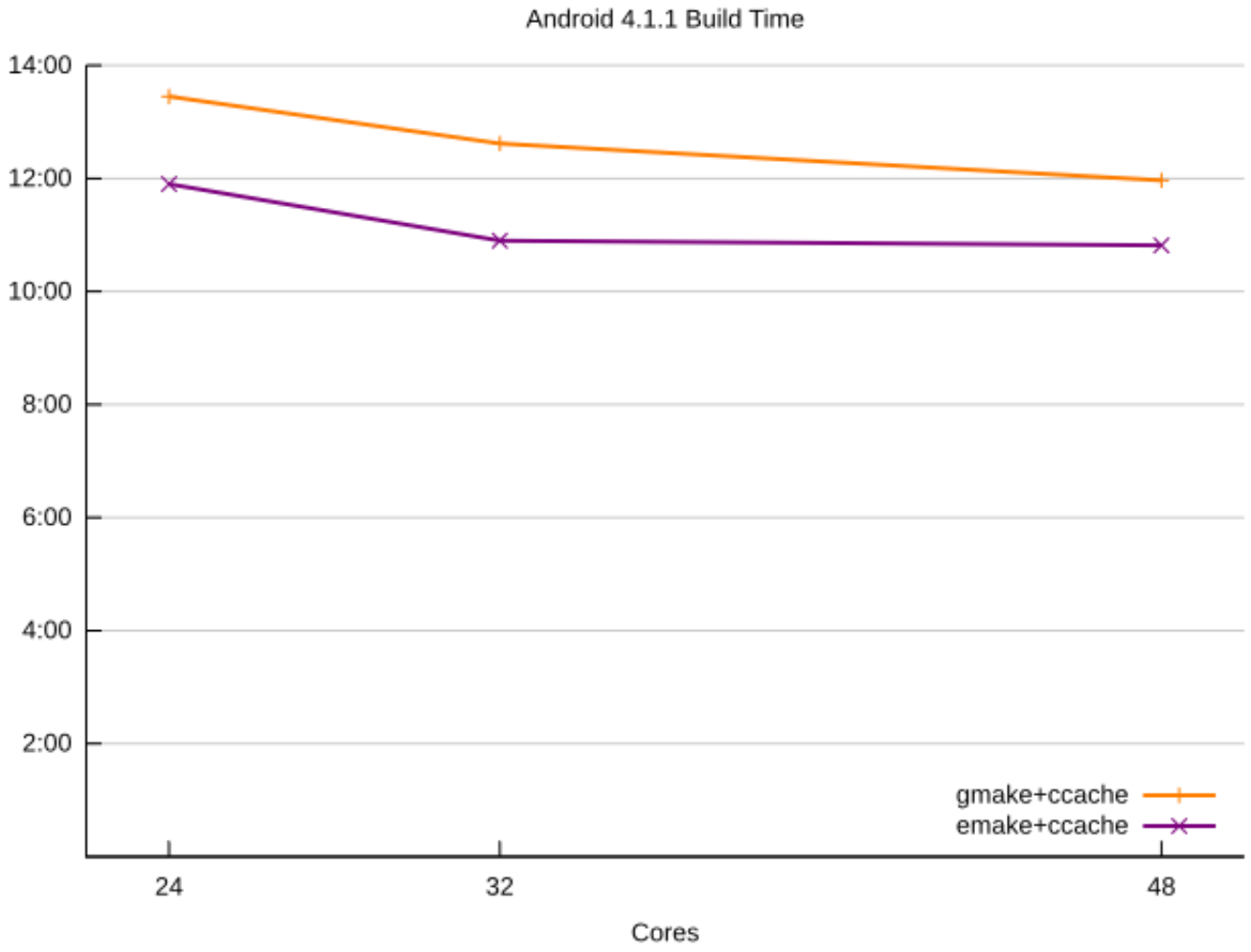
- Google has done a great job of optimizing Android builds
- But there's still room for improvement!



electric CLOUD

Summary – what about ccache?

- ccache complements other features for even faster builds



electric CLOUD

Availability

- ElectricAccelerator 7.0
  - Available late Q1 2013
  - Includes parse avoidance, dependency optimization

- Download from http://www.electric-cloud.com/eade

electric CLOUD

Q & A