Chromium updates from Arm

Higher, faster, stronger

Dave Rodgman & Adenilson Cavalcanti
Chromium team @Arm
2019-10-14

Intro: a talk in three parts

- New platforms
  - Bringing the web to Windows-on-Arm
- Faster
  - How we've made the web faster
- Stronger
  - Up-coming hardware security features that will make the web more secure

Bringing the web to new platforms

Windows-on-Arm

Bringing the web to Windows on Arm

- Lots of new devices emerging
  - Outstanding on power
    - Great battery life
    - Fanless (great form factor)
  - Strong performance
    - Native performance is excellent
    - But lots of software is x86 emulated

- Emulation works against performance, stability and power

- We've worked on enabling native builds of Chromium, Electron and CEF

- Chromium
  - Builds out-of-the-box for Windows on Arm
  - Performance is 2.4x faster on Speedometer
  - Stable – unit tests near parity with Intel

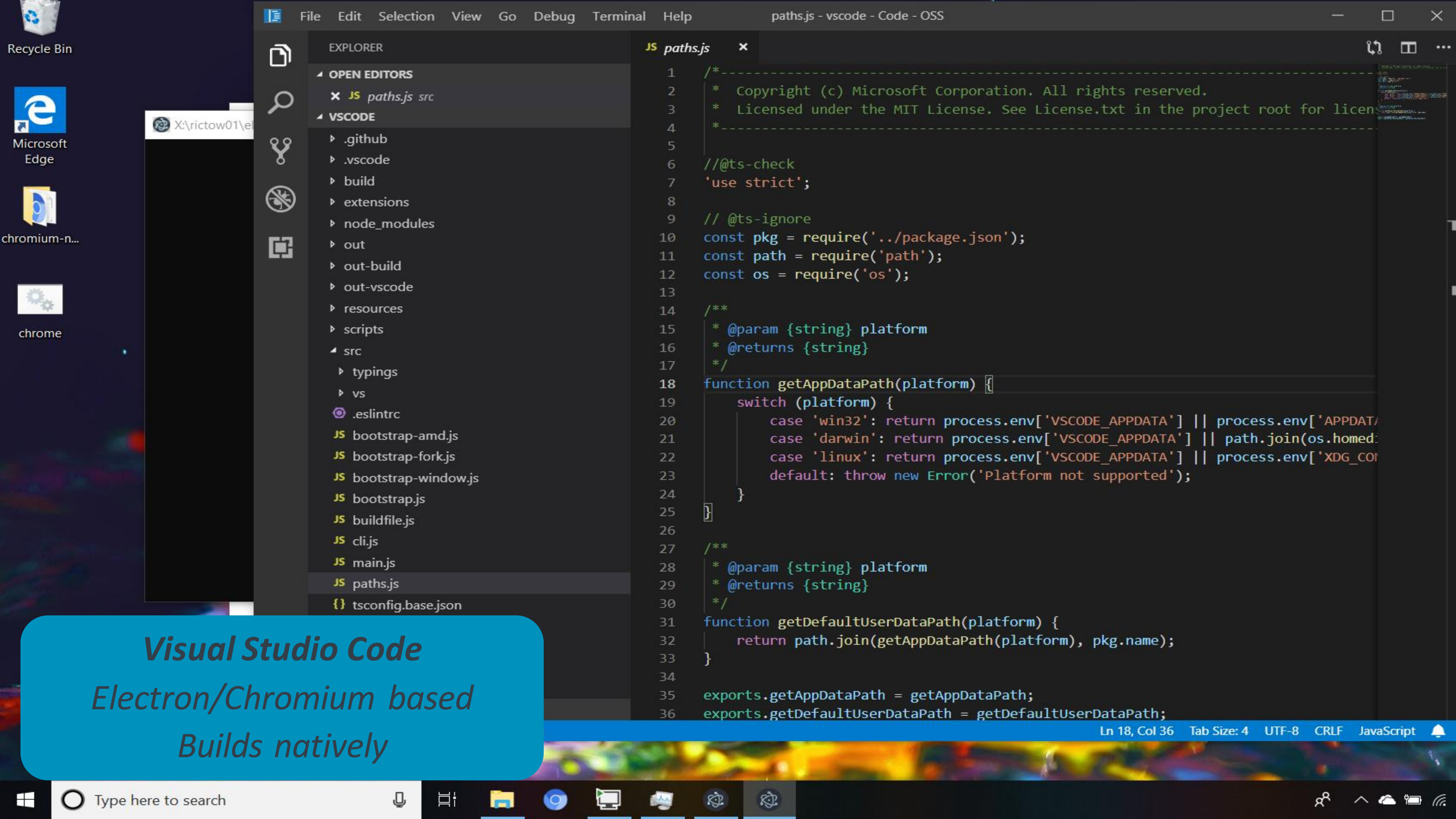
- Electron
  - Electron 6 has native support
  - We ported Visual Studio Code to prove the concept

- CEF
  - CEF 78 has native support
  - Unit test parity with Intel

Chromium 74+

2.4 times faster when native

arm



Visual Studio Code

Electron/Chromium based

Builds natively

Optimizing the browser for Arm

Android, ChromeOS, Windows, Linux

zlib: back in 2017's Hackfest (2 years ago...)

What comes next

- Land the libpng optimization.
- CRC32: ARMv8 instruction is about 10x faster.
- Fix infback corner case.
- Compression comes next.

Zlib users should consider migrating to Chromium's zlib.

Chromium's zlib status in 2019

- Land the libpng optimization. **Done*!**
- CRC32: Armv8 instruction is about ~~10x~~ 20x faster. **Done!**
- Fix infback corner case. **Done!**
- Compression comes next. **Done!**

* libpng patch.

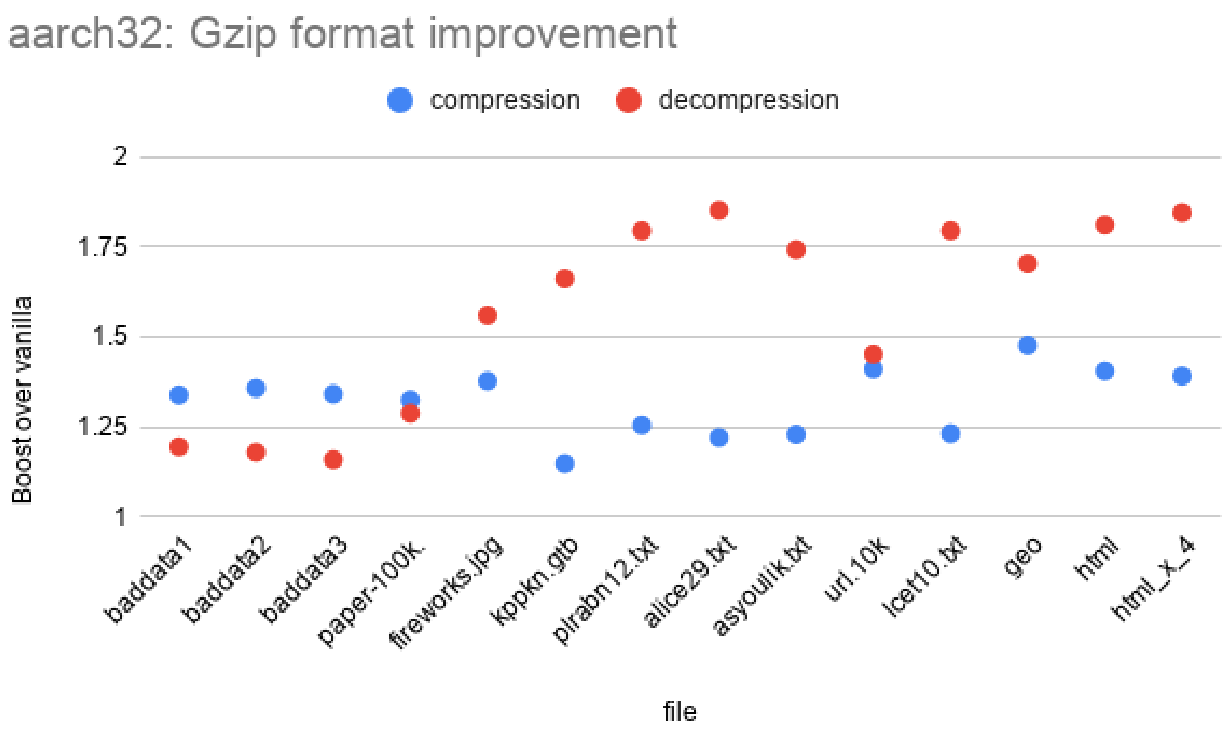
Chromium's zlib status in 2019

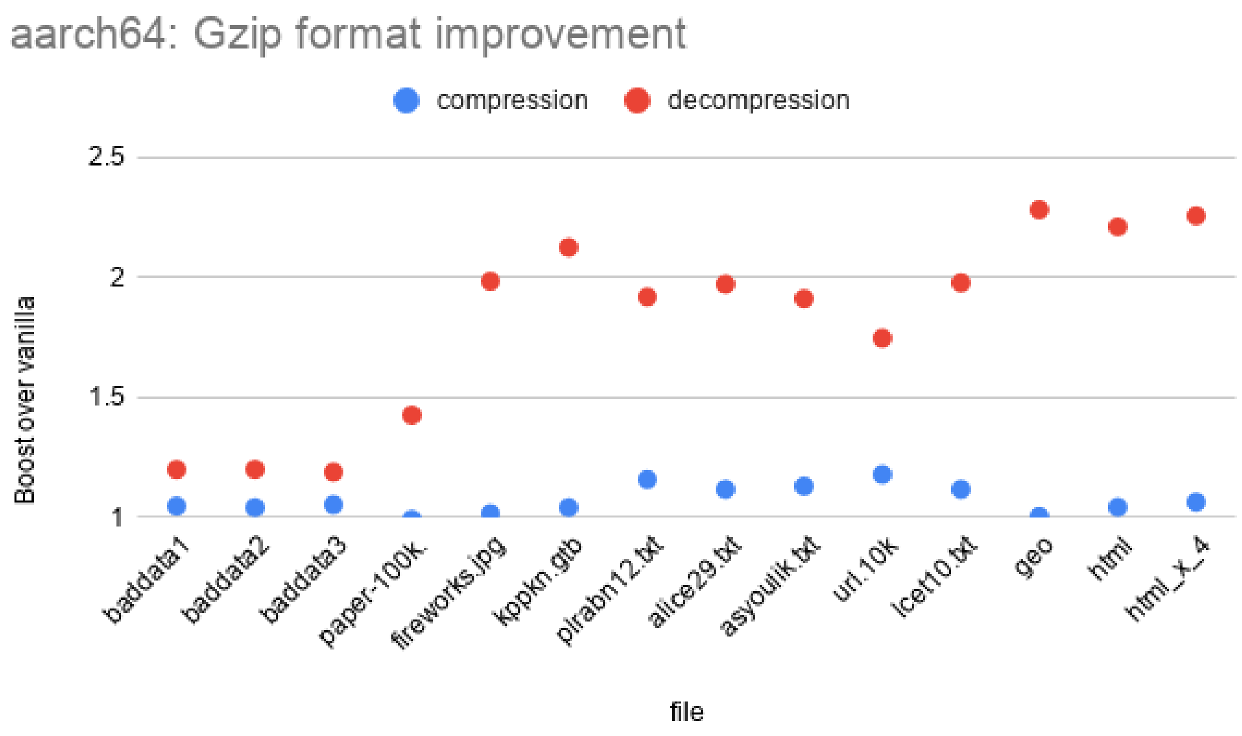
- Land the libpng optimization. **Done*!**
- CRC32: Armv8 instruction is about ~~10x~~ 20x faster. **Done!**
- Fix infback corner case. **Done!**
- Compression comes next. **Done!**
- **Android migration to Chromium's zlib. WIP**.**

* libpng patch.
** Android repo: https://android.googlesource.com/platform/external/zlib

Chromium's zlib: performance in 2019

Decompression (+70% to 120%), compression (+10% to 36%)





*Current data spreadsheet.

zlib: where is it used?

- Network operations (i.e. content-encoding: gzip)
- PNG decoding
- Chronet
  - (Chromium network library – used in other projects, e.g. Gsuite apps)

Optimizations enabled new use-cases

- **Javascript source strings compression**
- **V8 snapshots**
- **Chromium JS/HTML resource.pak**
- **Android: all things gzip (e.g. apks, java compression API, etc)**

JPEG: Optimizing Chromium's libjpeg-turbo

Work in progress... crbug/922430

Landed

Optimized paths for decompression

- Color conversion
- Upsampling (simple, fancy, merged)
- Inverse discrete cosine transform (scaling and regular)
- Average 24% reduced decode time (64-bit)

Optimizations written using NEON intrinsics

- Common source code implementation for both 32- and 64-bit
- Easier to maintain than assembly
- Allows security tools (such as MSan) to analyze optimized paths

Still to come

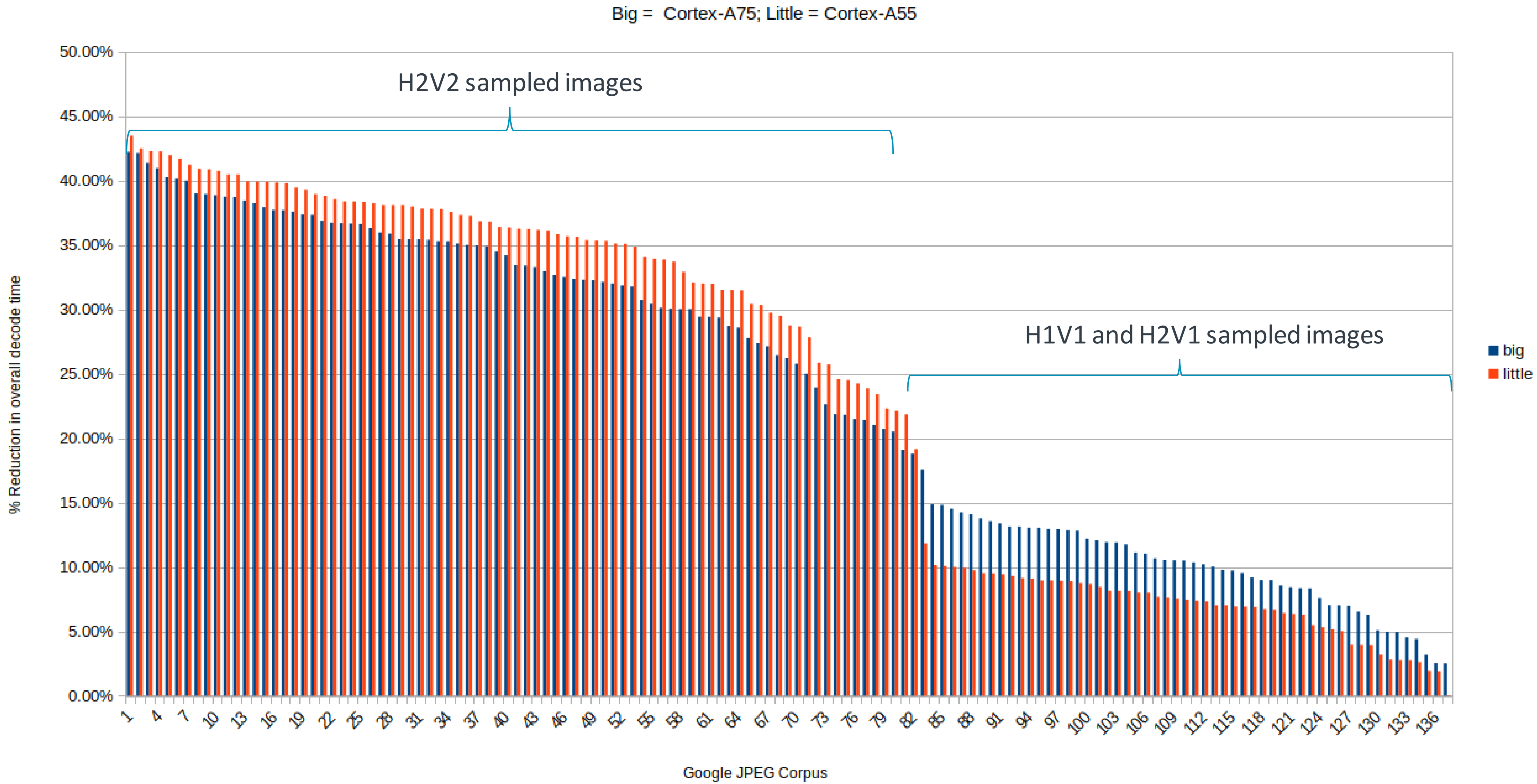
Push compression optimization patches

- Color conversion
- Downsampling
- Sample conversion / quantization
- Forward discrete cosine transform
- Huffman encoding

Push all optimizations to upstream project:

- Firefox and Safari will benefit too!

JPEG decode time reduction in Chrome (64-bit)



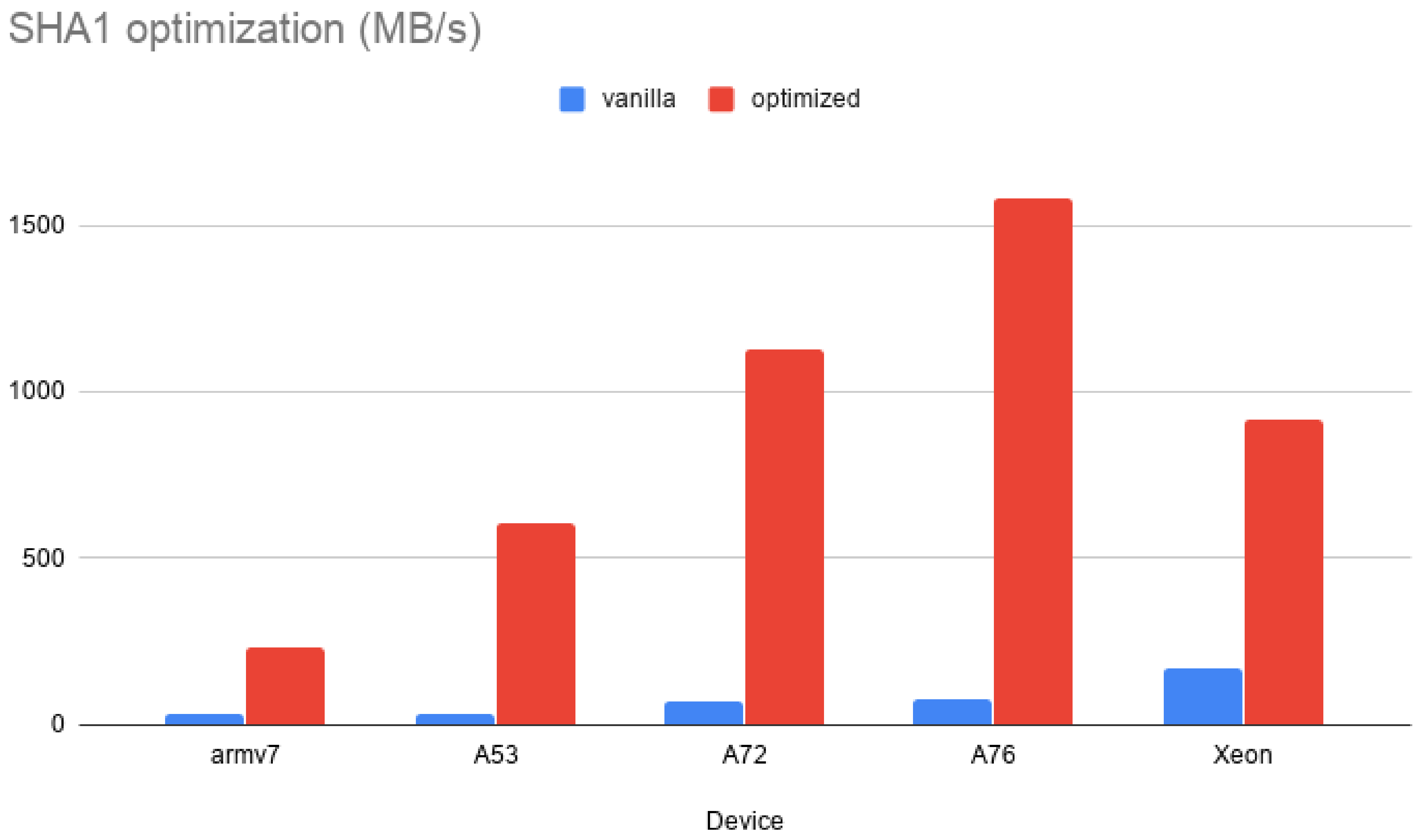
Hashing

- Started in ShapeCache: layout boost of 19%@x86 and 23%@Arm
- Improving hashing in Blink: 16x to 21x faster SHA1 leveraging BoringSSL, 7x faster SHA256
- Reduces time spent in v8.execute()*
- Ongoing effort to migrate code base from SuperFastHash() to FastHash**()

* 9% on wikipedia (265ms vs 289ms)

** Boost in 7x to 8x

Hashing: SHA1 boost between 6x (Intel) to 22x (Arm little cores)



FastHash(): initial experimental data

Find Histogram name | chromium@4246b31 | α=0.01 | %Δavg | Export | Show all | Help | Feedback

name | deviceIds | stories | benchmarkStart

Name	chromium@4246b31	chromium@4246b31 + 31283f1
many-block-children-auto-inline-size	55.139	+7.9%
attach-inlines	8.330	+7.1%
fixed-grid-lots-of-stretched-data	110.853	+7.1%
flexbox-row-stretch-height-definite	75.216	+7.0%
many-block-children-fixed-inline-size	58.011	+6.0%
fit-content-change-available-size-blocks	11.474	+5.5%
flexbox-deeply-nested-column-flow	1,274.122	+4.1%
nested-blocks-with-percent-height-and-…	884.591	+4.0%
change-text-css-contain	6.036	+3.8%
line-layout-repeat-append-select	21.411	+2.9%
multicol_fixed-height-with-spanner-and-…	45.298	+2.3%
auto-grid-lots-of-data	60.588	+2.3%
line-layout-repeat-append	28.046	+2.2%
flexbox-with-stretch-layout	3.372	+2.0%
fixed-grid-lots-of-data	117.390	+1.9%
multicol_tall-content-short-columns-reali…	28.227	+1.9%
large-grid	0.345	+1.8%
attach-inlines-2	1,569.412	+1.7%
large-spanning-grid-item	278.828	+1.7%
flexbox-row-wrap	222.199	+1.6%
flexbox-column-nowrap	509.937	+1.5%
flexbox-column-wrap	505.971	+1.4%
nested-percent-height-tables	0.055	+1.1%
multicol_deeply-nested-tables	2,477.791	+1.0%
flexbox-row-nowrap	225.099	+0.9%
character_fallback	2,873.948 ms	+0.8%
add-remove-inline-floats	7.934	+0.8%
line-layout	25.046	+0.7%

Pinpoint: https://pinpoint-dot-chromeperf.appspot.com/job/14d4ef83f40000

Hashing: next steps

- Migrate code base to use the faster hash (i.e. 3GB/s to 15GB/s @Intel, similar boost for Arm).
- Slow progress: legacy code (e.g. UMA metrics).
- Tracking bug: https://bugs.chromium.org/p/chromium/issues/detail?id=902789#c31

Harfbuzz: minor changes

Optimizations

- Experimented with branchless binary search
  - faster on little cores… but slower for big cores
  - tried various approaches, sometimes faster on big, but never both at once :-(

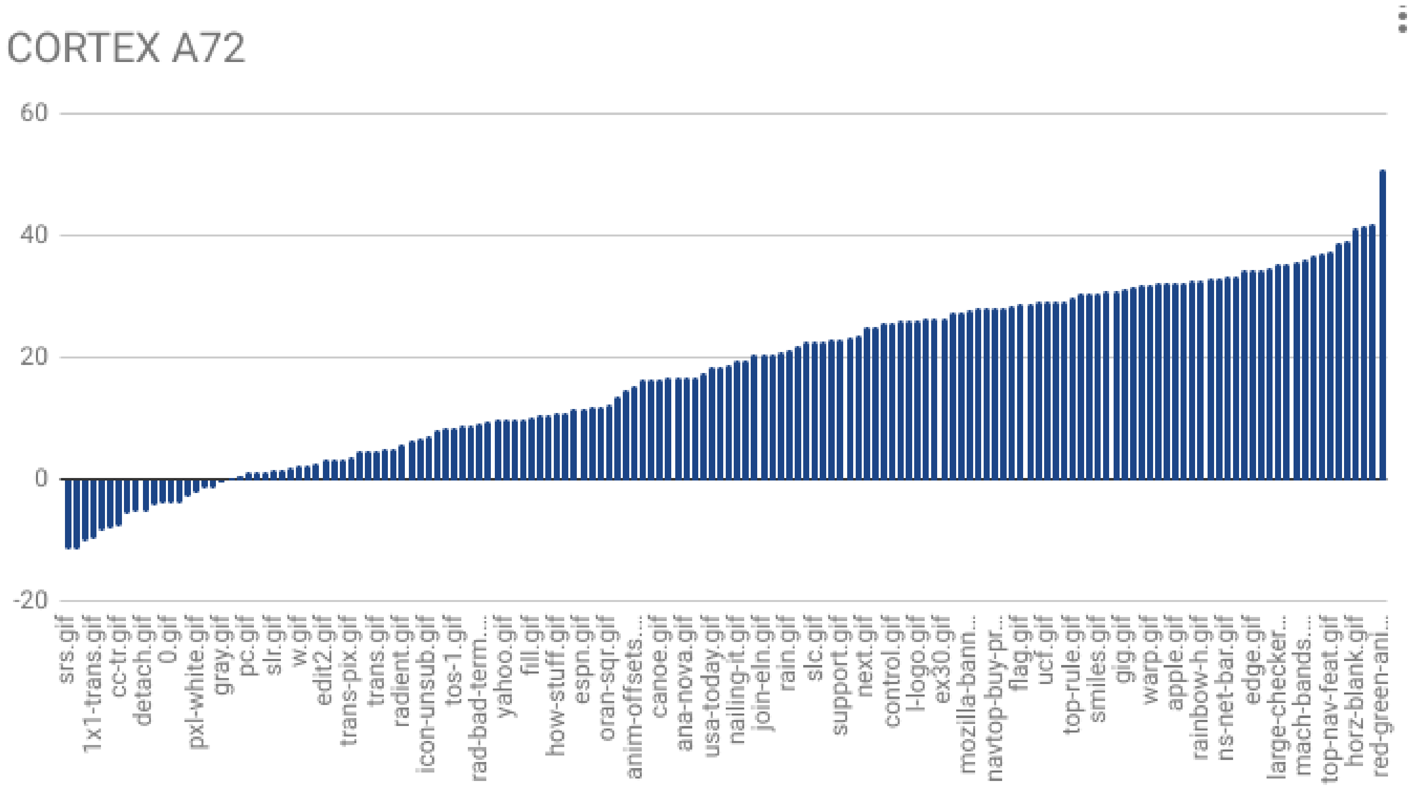
- Faster big endian conversion (using REV16 / BSWAP instruction)
  - 8 - 11% boost for Latin

- Compiler optimizations:
  - 9 - 10% additional for Latin
  - Take-away: -O3 makes a difference!

GIF

GIF decode

- Restructured LZW decoding.
- Write output in larger (8 byte) chunks.
- Regressions are for very small images so low impact.
- Average 17% improvement on big cores.



New architectural features for a more secure web

MTE, PAuth, BTI

Security from the CPU upwards

Coming soon: hardware features which secure the web

- Securing the web means working at lots of levels
  - JS frameworks, browser APIs, etc
  - But also low levels – hardware features can boost security
  - Browser is all about handling untrusted data

- Common attack vectors include
  - Memory bugs
  - Control Flow Integrity

2/3 of Critical Vulnerabilities & Exposures (CVEs)

- Recent iterations of the Arm architecture introduced lots of security-related features
  - **Pointer Authentication**
  - **Memory Tagging**
  - **Branch Target Identification**
  - Crypto Extensions
  - Random Number Generator

Memory Tagging Extensions

Detect common memory bugs

- Upper bits in pointers store ‘color’ for allocated data
- Memory also records its color (each 16-byte block has a 4-bit color)
- Non-invasive
  - Only the memory allocator is changed
- Protect against
  - Buffer overflow
  - Use-after-free
- Precise vs imprecise
  - Precise: synchronous, slower
  - Imprecise: asynchronous, don’t capture exact location, low overhead

```
char *ptr = new char[16]; // memory colored
```

```
ptr[17] = 5;  // color mismatch -> overflow
```

```
delete[] ptr; // memory re-colored on free
```

```
ptr[10] = 10; // color mismatch -> use-after-free
```

Memory Tagging Extensions usage models & benefits

Detection

Browser in the field

- **Imprecise checking** will detect any tagging mismatch
- May be used for all **applications** and performance sensitive processes
- Targeting minimal performance overhead

OEMs/OSV get in-field issue reports before they are turned into exploits – even in 3rd party code and apps

Mitigation

Privileged Software

Key system services

- **Precise checking** will stop execution on a tag mismatch
- Can be used for **system services** and other privilege escalation targets
- Some performance overhead is expected

OEMs/OSV protect their consumers data by aborting as soon as a vulnerability in critical services is detected

Debug

Browser debugging

- **Precise checking** will stop execution on a tag mismatch
- Can be used for all SW during pre-production cycles
- Some performance overhead is expected

OEMs/OSV find errors before they ever leave the factory reducing support costs and damage to reputation

Memory Tagging Extensions

Internal prototyping in Chromium

Oilpan

- Prototype designed, implemented, tested on a model
  - Testing on a model works well
  - Changes were limited to the allocator – no impact on the rest of the codebase
- Passes all blink_heap_unittests
- Some memory size increase (object headers increase; object size must be multiple of 16 bytes)
- Overall, memory increases by 1% (which causes around 1% performance drop)
- Potentially some additional performance overhead from tagging and checking

libjpeg-turbo

- Good candidate for buffer-overflow attacks via malformed JPEGs
- Nested allocation
  - Uses its own front-end allocator on top of the heap manager (jemalloc)
  - Front-end receives a large tagged block
  - Front-end re-tags sub-blocks on allocation
- Very small code changes were needed

Prototypes confirmed MTE is non-invasive and low-impact

Pointer Authentication

Defend against ROP attacks

- What's a gadget?
  - Attacker uses (e.g.) buffer over-run to corrupt stack
  - Return address replaced with address of a *gadget*
  - *Gadget* is a pre-existing short sequence of instructions (followed by return)
  - After gadget executes & returns, the next gadget runs, etc
  - A sequence of gadgets is Turing complete

- How Pointer Authentication can help
  - Return address is cryptographically signed (not encrypted) on function entry
    - Return address is combined with stack pointer, and signed with a secret key
    - Use of stack pointer helps protect against pointer substitution
  - Immediately before returning, pointer is validated and signature removed

- Function prologue:

```
PACIASP
SUB         sp, sp, #0x40
STP         x29, x30, [sp,#0x30]
ADD         x29, sp, #0x30
```

Sign return address (modify x30)

- Function epilogue:

```
LDP         x29,x30,[sp,#0x30]
ADD         sp,sp,#0x40
AUTIASP
RET
```

Validate & remove signature (from x30)

Illegal address exception if validation failed

More details: https://ourcommunity.arm.com/forums/index.php?/blog/446/entry-3195-pointer-authentication-basics/

arm

Pointer Authentication

Impact on the browser

- What does it defend against?
  - Malicious input data
    - primarily Javascript
    - also malformed images, etc
  - Prevent attackers from executing arbitrary code
  - Web browsers have potential for high benefit [1]
- Just a compiler flag?
  - Almost, but some impact in stack unwinding (e.g. libunwind)
  - JIT compilers should generate PAuth instructions in order to benefit
  - Otherwise minimal impact

- Binary compatibility
  - Encoded in NOP space
  - Old hardware ignores PAuth instructions
- Performance impact
  - Two additional instructions per function
    - Reduce overhead by skipping leaf functions
  - Expect less than 2% performance overhead
- Future: data pointer authentication
  - Instructions exist for authenticated memory reads
  - Could offer additional protection at higher overhead

[1] https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

Branch Target Identification

Complement to Pointer Authentication

- Pointer Authentication
  - Secures the control flow between functions
  - Limits attacker's ability to *call* gadgets
- Branch Target Identifcation
  - Protects the targets of indirect branches
  - Restricts *what can be used as a gadget*
- Mechanism
  - Introduce new BTI instruction (aka "landing pad")
  - Indirect branches may only branch to BTI instructions
  - Just a compiler flag
    - (assembly may need manual BTI instructions)
- Binary compatibility
  - Encoded in NOP space
  - Old hardware ignores BTI instructions

Combined Benefit

glibc (Ubuntu 14.04) gadgets ≤ 10 instructions

- Longer gadgets are less useful

Type	No PAC/BTI	With PAC/BTI	Reduction
ROP	16,585	264	59x
JOP	5,845	11	531x
Combined	**22,430**	**275**	**82x**

What does this mean for the browser?

More secure, most code needs no change

Work needed: non-invasive

- **Memory Tagging Extensions**
  - Allocators (and de-allocators) need changes to handle tagging
  - Lots of options to trade-off performance vs. level of benefit
  - Bulk of code will not change
- **Pointer Authentication**
  - Stack unwinding code needs minor changes
- **Branch Target Identification**
  - Assembly code may need to add BTI instructions

Benefits

- Massive reduction in ROP/JOP attack surface
  - Defend against malicious input
- Detect most common type of CVE (memory bugs)
  - Spot security issues **before they are exploited**

Questions

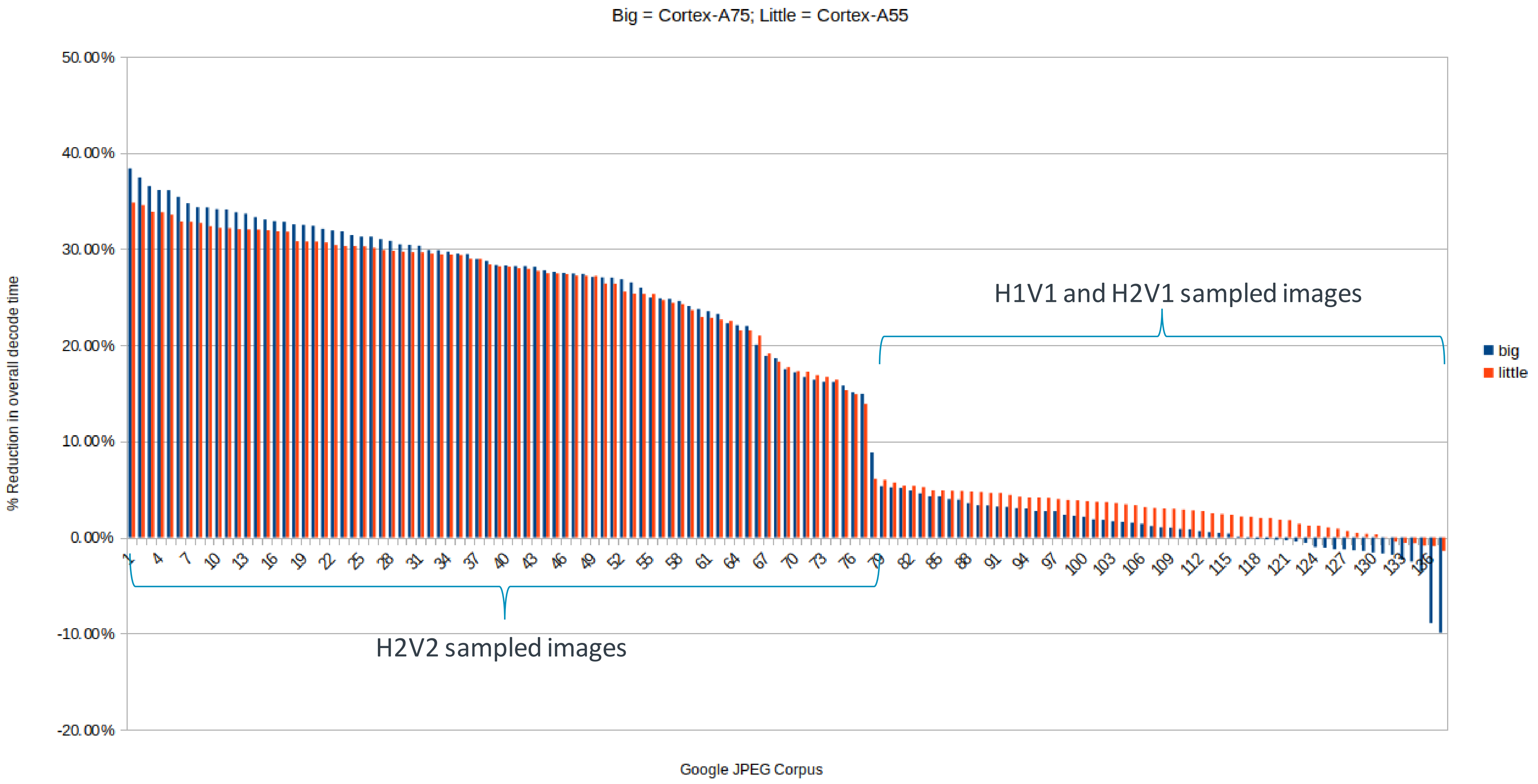
- Are there areas that would benefit from optimisation on Arm?
- What should Arm be doing to help the browser?
    - Performance
    - Security
    - Other

Chromium team @Arm

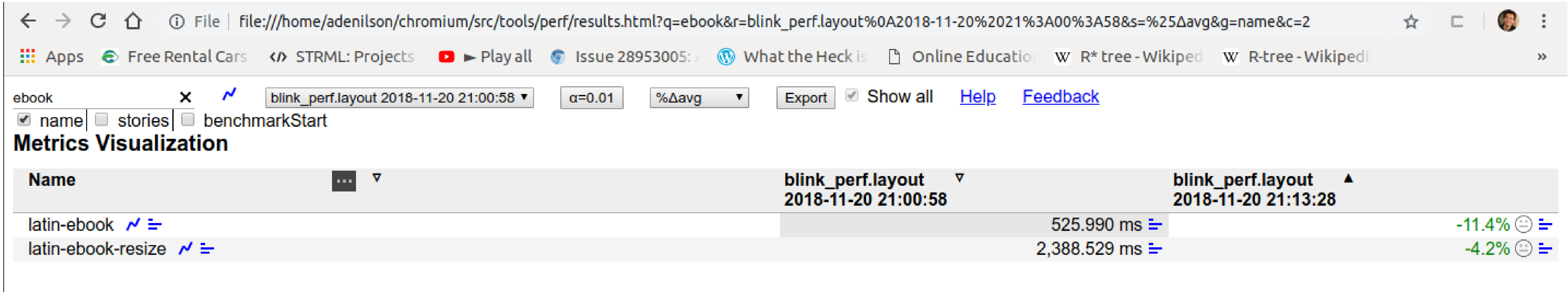
dave.rodgman@arm.com
adenilson.cavalcanti@arm.com
stephen.kyle@arm.com
richard.townsend@arm.com
jonathan.wright@arm.com
andrea.brunato@arm.com
jack.davison@arm.com

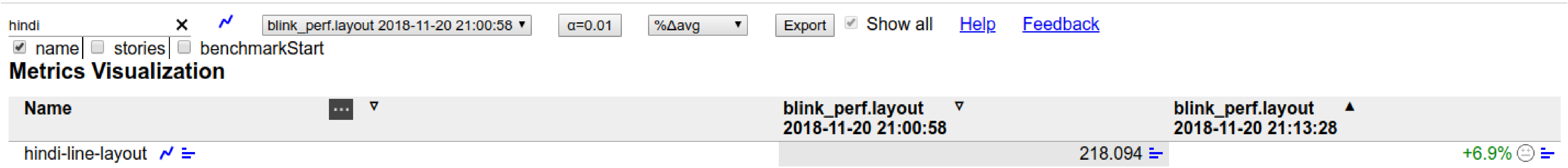
Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה

JPEG decode time reduction in Chrome (32-bit)



Harfbuzz: byte swapping





Miscellaneous optimizations

- Helping out with V8's pointer compression implementation on AArch64
  - improved performance of Speedometer by 1%
- Instruction scheduling improvements for V8 builtins
  - improved other JS benchmarks (octane, jetstream, etc) by 1.2%
- Some improvements for brotli as well
  - 5% reduction in decompression time on little cores