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
Visual Studio Code
Electron/Chromium based
Builds natively
Optimizing the browser for Arm

Android, ChromeOS, Windows, Linux
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**!


© 2019 Arm Limited
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](https://android.googlesource.com/platform/external/zlib)

** Android repo: [https://android.googlesource.com/platform/external/zlib](https://android.googlesource.com/platform/external/zlib)
Chromium’s zlib: performance in 2019

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

*aCurrent 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](https://crbug.com/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)

**Still to come**

Push compression optimization patches
- Color conversion
- Downsampling
- Sample conversion / quantization
- Forward discrete cosine transform
- Huffman encoding

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

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)
<table>
<thead>
<tr>
<th>Name</th>
<th>chromium@4246b31</th>
<th>chromium@4246b31 + 31283f1</th>
</tr>
</thead>
<tbody>
<tr>
<td>many-block-children-auto-inline-size</td>
<td>55.139</td>
<td>+7.9%</td>
</tr>
<tr>
<td>attach-inlines</td>
<td>8.330</td>
<td>+7.1%</td>
</tr>
<tr>
<td>fixed-grid-lots-of-stretched-data</td>
<td>110.853</td>
<td>+7.0%</td>
</tr>
<tr>
<td>flexbox-row-stretch-height-define</td>
<td>75.216</td>
<td>+6.0%</td>
</tr>
<tr>
<td>many-block-children-fixed-inline-size</td>
<td>58.011</td>
<td>+5.5%</td>
</tr>
<tr>
<td>fit-content-change-available-size-blocks</td>
<td>11.474</td>
<td>+4.1%</td>
</tr>
<tr>
<td>flexbox-deeply-nested-column-flow</td>
<td>1,274.122</td>
<td>+4.0%</td>
</tr>
<tr>
<td>nested-blocks-with-percent-height-and-...</td>
<td>884.591</td>
<td>+3.8%</td>
</tr>
<tr>
<td>change-text-css-contains</td>
<td>6.036</td>
<td>+2.9%</td>
</tr>
<tr>
<td>line-layout-repeat-append-select</td>
<td>21.411</td>
<td>+2.3%</td>
</tr>
<tr>
<td>multicol_fixed-height-with-spanner-and-...</td>
<td>45.298</td>
<td>+2.3%</td>
</tr>
<tr>
<td>auto-grid-lots-of-data</td>
<td>60.588</td>
<td>+2.3%</td>
</tr>
<tr>
<td>line-layout-repeat-append</td>
<td>28.048</td>
<td>+2.2%</td>
</tr>
<tr>
<td>flexbox-with-stretch-layout</td>
<td>3.372</td>
<td>+2.0%</td>
</tr>
<tr>
<td>fixed-grid-lots-of-data</td>
<td>117.390</td>
<td>+1.9%</td>
</tr>
<tr>
<td>multicol_tall-content-short-columns-reall...</td>
<td>28.227</td>
<td>+1.8%</td>
</tr>
<tr>
<td>large-grid</td>
<td>0.345</td>
<td>+1.7%</td>
</tr>
<tr>
<td>attach-inlines-2</td>
<td>1,569.412</td>
<td>+1.7%</td>
</tr>
<tr>
<td>large-spanning-grid-item</td>
<td>278.828</td>
<td>+1.6%</td>
</tr>
<tr>
<td>flexbox-row-wrap</td>
<td>222.199</td>
<td>+1.5%</td>
</tr>
<tr>
<td>flexbox-column-noswap</td>
<td>509.937</td>
<td>+1.4%</td>
</tr>
<tr>
<td>flexbox-column-wrap</td>
<td>505.971</td>
<td>+1.4%</td>
</tr>
<tr>
<td>nested-percent-height-tables</td>
<td>0.055</td>
<td>+1.1%</td>
</tr>
<tr>
<td>multicol_deepest-nested-tables</td>
<td>2,477.791</td>
<td>+1.0%</td>
</tr>
<tr>
<td>flexbox-row-noswap</td>
<td>225.099</td>
<td>+0.9%</td>
</tr>
<tr>
<td>character_fallback</td>
<td>2,873.948 ms</td>
<td>+0.8%</td>
</tr>
<tr>
<td>add-remove-inline-floats</td>
<td>7.934</td>
<td>+0.8%</td>
</tr>
<tr>
<td>line-layout</td>
<td>25.046</td>
<td>+0.7%</td>
</tr>
</tbody>
</table>

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

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

2/3 of Critical Vulnerabilities & Exposures (CVEs)
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

```c
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

---

## 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

---

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

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

**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

- Function epilogue:
  LDP x29,x30,[sp,#0x30]
  ADD sp,sp,#0x40
  AUTIASP
  RET

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

Branch Target Identification
Complement to Pointer Authentication

- Pointer Authentication
  - Secures the control flow between functions
  - Limits attacker’s ability to call gadgets

- Branch Target Identification
  - 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

<table>
<thead>
<tr>
<th>Type</th>
<th>No PAC/BTI</th>
<th>With PAC/BTI</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP</td>
<td>16,585</td>
<td>264</td>
<td>59x</td>
</tr>
<tr>
<td>JOP</td>
<td>5,845</td>
<td>11</td>
<td>531x</td>
</tr>
<tr>
<td>Combined</td>
<td>22,430</td>
<td>275</td>
<td>82x</td>
</tr>
</tbody>
</table>
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
JPEG decode time reduction in Chrome (32-bit)
## Harfbuzz: byte swapping

### Metrics Visualization

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>latin-ebook-resize</td>
<td>525.990 ms</td>
<td>2,388.529 ms</td>
</tr>
<tr>
<td>hindi-line-layout</td>
<td>2018-11-20 21:00:58</td>
<td>218.094 ms</td>
</tr>
</tbody>
</table>
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