Part 1

WebKit JIT engine

GCC in action (x64)

```
$ cat func01.c

int func01(int a, int b) {

   return a + b;

}

$ gcc -c func01.c -o
func01.o

$ objdump -d --no-show-raw-
insn func01.o
```

```
…

<func01>:

   0: push   %rbp

   1: mov    %rsp,%rbp

   4: mov    %edi,-0x4(%rbp)

   7: mov    %esi,-0x8(%rbp)

   a: mov    -0x4(%rbp),%edx

   d: mov    -0x8(%rbp),%eax

  10: add    %edx,%eax

  12: pop    %rbp

  13: retq
```

GCC in action (MIPS)

```
$ cat func01.c

int func01(int a, int b) {

   return a + b;

}

$ mipsel-linux-gcc -c
func01.c -o func01.o

$ mipsel-linux-objdump -d
--no-show-raw-insn func01.o
```

```
<func01>:

   0: addiu sp,sp,-8

   4: sw s8,4(sp)

   8: move  s8,sp

   c: sw a0,8(s8)

  10: sw a1,12(s8)

  14: lw v1,8(s8)

  18: lw v0,12(s8)

  1c: addu  v0,v1,v0

  20: move  sp,s8

  24: lw s8,4(sp)

  28: addiu sp,sp,8

  2c: jr ra

  30: nop
```

JIT levels

- LLint: representation of JavaScript in platform independent byte code
- BaseLine JIT: first (non-optimized) generated
- DFG (Data Flow Graph) JIT: optimized code
- *FTL JIT (x64 only): improved optimized code*
  - *Not available to us on DAWN, ignored here*

- Often executed functions will be optimized

Controlling JIT levels

- Environment vars controlling JIT behavior:
  - JSC_enableJIT (“true”/”false”, default: “true”), if disabled, only LLint is used
  - JSC_enableDFGJIT: if disabled, only use baseline JIT
  - JSC_showDisassembly: whenever function is JITted, print info to stderr
  - JSC_thresholdForJITSoon (default: 100): how many points does a function need to have before being JITted
    - Function scores points when called or when containing loop

More JIT environment variables

- JSC_reportBaselineCompileTimes: writes compile times to stderr
- JSC_bytecodeRangeToJITCompile=N:M : sets size range of function to compile to baseline
- JSC_bytecodeRangeToDFGCompile=N:M : sets size range of function to compile by DFG
- JSC_jitWhitelist=<filename> : only JIT functions listed in white list file

Where are options defined?

- Check:
  - Source/JavaScriptCore/runtime/Options.h
- v(int32, thresholdForJITSoon, 100)
  - Variable type, name (without “JSC_” prefix) and default value

- Macro magic is interesting in its own right and is worth a read

Containers eat stderr

- On DAWN qtbrowser runs in a container
- Even just storing stderr stream in file is complicated
  - We (including PACE) gave up
- Instead, webkit allows for writing everything to file:
  - “Source/WTF/wtf/Datalog.cpp”:
    - #define DATA_LOG_TO_FILE 1
    - #define DATA_LOG_FILENAME "/tmp/WTFLog"
- Syslog wasn't an option for data-heavy logging

HTML JITting example

```
<script>
var globalVar = 0;
var callCount = 100;
function func01(addedValue) {
   globalVar += addedValue;
}
function triggerJIT() {
   for (i = 0; i < callCount; i++) {
      func01(i);
   }
}
</script>
<button onclick="triggerJIT()">Trigger JIT</button>
```

Inspect generated code

- Run with “JSC_reportBaselineCompileTimes=true”
  - This will give the full name of a function
- Add name to white list file
- Run with:
  - “JSC_jitWhitelist=$PWD/whitelist.txt”
  - “JSC_showDisassembly=true”
- Look for:
  - “Code at [*start address*, *end address*):”
- Extract code from core file, or via attached gdb(server)

DEMO

Instruction count

- Simple function has many instructions
- Two reasons:
  - Code is not optimized
  - JavaScript is dynamically typed
- Example:

```
function add(a, b) { return a + b; }
```

  - Works for ints, but also strings, floats, lists, etc...
- Generated code needs to deal with that

Part 2

Our role as integrators

Division of labor

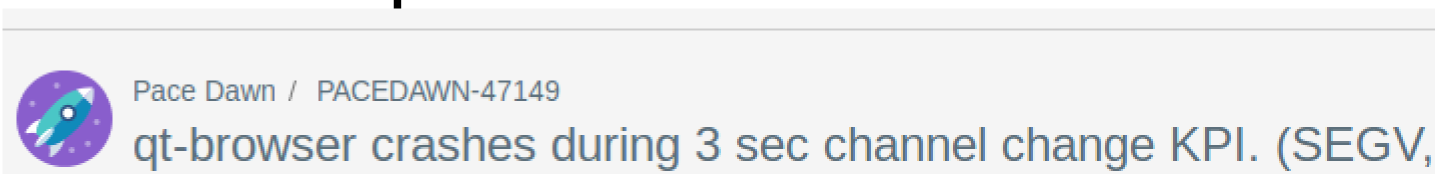
"Resources from Metro can be divided into *developers* who work on generic browser bugs, and *integrators* who solve platform-specific issues.

Yet...

- Some things are indeed very platform specific:
  - PACE SDK
  - Deploying custom build of browser
  - Traxis etc..
- In practice, of course, the line is less clear
- This JIT issue, for example, only happens in DAWN..., so?

Steps in resolving the issue

1. Someone posted issue on Jira:



2. Attached is a core file, so we had to build PACE SDK
3. As it turned out, qtwebkit didn't have symbols in so-file
4. It turned out to be impossible (at least very difficult) to have qmake leave all the symbols in

Continued...

5. Eventually opted for overwriting PACE build with scripts that replace args to gcc
6. Crash location seemed to be outside of regular C++ code
7. Modified arguments to disable optimization
8. This new build was deployed in test environment
9. Issue turned out hard to be reproduce, because it took a long time, and NAF often caused OoMs
10. Once we had the corefile, location still outside of regular code

Continued...

11. Two possible explanations were considered: bug in JIT, or bug in GCC
    - GCC is patched by broadcom, had issues before
    - We disabled baseline JIT → issue disappeared (*very likely* JIT)
12. Enabled JSC_showDissassembly
13. Found out how to write to log (owner rights are tricky in containers)
14. Issue disappeared, constantly, but so did performance
15. Received patch from Guillaume to log less

Continued...

16. Patch turned out to be for WPE, had to be back ported
17. This resulted in large (>25GB) files containing function names and their location
    - Couldn't just focus on last bit, because function could have been JITted hour before
18. Wrote script to parse this file
19. Crash site appeared to be in JITted function
    - But code looked no way like expected and couldn't explain crash

GDB

- As mentioned, assembly code in corefile doesn't link up with C++ code
  - Wasn't a compile issue
  - JIT just generates code, doesn't introduce symbols
- Opening corefile in GDB won't allow you to do backtraces, but there are other useful commands

GDB commands

- `info registers`: prints contents of registers at time of crash
  - Important: PC (program counter) and RA (return address)
- `x/<count>i <addr>`: prints contents of memory as <count> instructions (disassembles)
- `set auto-solib-add off`: don't load symbols from so-files (saves time + potential crash)

Our issue: GDB session

```
Program terminated with signal
SIGSEGV

#0  0x40000000 in ?? ()

(gdb) bt

warning: GDB can't find the start
of the function at 0x4d1f3ca7.

...

#0  0x40000000 in ?? ()

#1  0x4d1f3ca8 in ?? ()

(gdb) info registers

      pc       ra

40000000 4d1f3ca8
```

```
(gdb)  x/4i 0x4d1f3ca0

   0x4d1f3ca0: jal 0x4d005460

   0x4d1f3ca4: nop

   0x4d1f3ca8: sw  v0,96(s0)

   0x4d1f3cac: sw  v1,100(s0)

(gdb) x/4i 0x4d005460

   0x4d005460: sw  zero,-8(s0)

   0x4d005464: lui t2,0x4cea

   0x4d005468: ori t2,t2,0x1994

   0x4d00546c: sw  s0,0(t2)
```

So... what was it?

- A CPU contains cache to improve access speed for often-used data and instructions
- The DAWN CPU has two separate caches for memory and instructions
- JIT writes to memory, so accesses memory cache
- CPU retrieves instructions via instruction cache
- Broadcom “Zephyr” architecture contains bug
- Kernel patch with forced syncs survives KPI

Bonus: calling convention

```
void func01(int * arg1,
            int * arg2,
            int * arg3)
{
   *arg1 = 1;
   *arg2 = 2;
   *arg3 = 3;
}
```

```
<func01>:
    0: movl    $0x1,(%rdi)
    6: movl    $0x2,(%rsi)
    c: movl    $0x3,(%rdx)
   12: retq
```

arg1->rdi, arg2->rsi, arg3->rdx

Bonus: volatile

```
$ cat normal.c
void func01(int * arg){
   int i = 0;
   *arg = 0;

   for (; i < 100; i++)
      *arg += i;
}
$ gcc -O3 -c normal.c -o normal.o && objdump -d normal.o
<func01>:
   0: movl   $0x1356,(%rdi)
   6: retq
```

arg1->rdi, arg2->rsi, arg3->rdx

volatile

```
$ cat volatile.c
void func01(volatile int * arg) {
    int i = 0;
    *arg = 0;

    for (; i < 100; i++)
        *arg += i;
}
```

- What code will be generated with “-O3” (full optimization)?

volatile

```c
void func01(volatile int * arg)
{
    int i = 0;
    *arg = 0;

    for (; i < 100; i++)
        *arg += i;
}
```

```
<func01>:
   0: movl   $0x0,(%rdi)
   6: xor    %eax,%eax
   8: nopl   0x0(%rax,%rax,1)
  10: mov    (%rdi),%edx
  12: add    %eax,%edx
  14: add    $0x1,%eax
  17: cmp    $0x64,%eax
  1a: mov    %edx,(%rdi)
  1c: jne    10 <func01+0x10>
  1e: repz retq
```

arg1->rdi, arg2->rsi, arg3->rdx

Bonus: `__restrict__`

```
$ cat normal.c
void func01(int * arg1 , int * arg2, int * arg3) {
   *arg1 += *arg3;
   *arg2 += *arg3;
}
$ gcc -O3 -c normal.c -o normal.o && objdump -d normal.o
<func01>:
   0: mov    (%rdx),%eax
   2: add    %eax,(%rdi)
   4: mov    (%rdx),%eax  ;<--- ???
   6: add    %eax,(%rsi)
   8: retq
```

arg1->rdi, arg2->rsi, arg3->rdx

__restrict__

```
$ cat restrict.c

void func01(int *__restrict__ arg1 , int *__restrict__ arg2, int
*__restrict__ arg3) {

    *arg1 += *arg3;

    *arg2 += *arg3;

}

$ gcc -O3 -c restrict.c -o restrict.o && objdump -d restrict.o

<func01>:

    0:  mov     (%rdx),%eax

    2:  add     %eax,(%rdi)

    4:  add     %eax,(%rsi)

    6:  retq
```

arg1->rdi, arg2->rsi, arg3->rdx