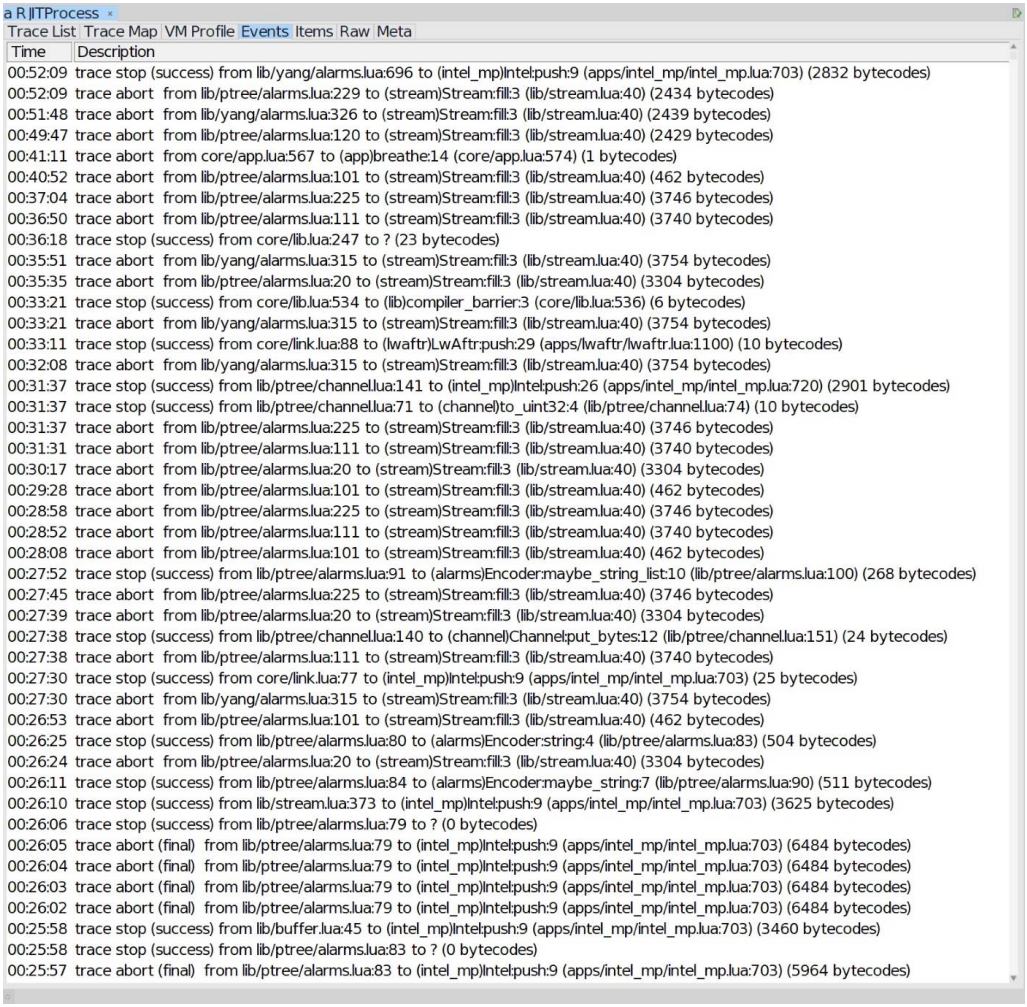
Hacking RaptorJIT: hot or not?

Max Rottenkolber <max@mr.gy>

Tuesday, 10 December 2019

Recently we investigated some latency fluctuations (i.e., packet loss) in Igalia's lwAFTR, a non-trivial Snabb application. I think the initial suspects were garbage collector pressure from stray heap allocations or a stray syscall blocking the engine. After ruling out GC and reaching for trusty old `strace` we could see... `mprotect` calls!

What calls `mprotect` to mark pages *read-write* and then calls `mprotect` again to mark them *read-execute*? A just-in-time compiler. Turns out RaptorJIT decided to compile new traces long after the application had started, inducing latency and causing packets to be dropped.



New traces compiled by RaptorJIT way after the application started to process packets.

Looking into the offending code paths, they appeared to be mostly periodic house keeping tasks. Things that are run in timers at regular intervals to do this and that, and other rarely taken branches. They were getting "hot" and being compiled, but should they?

My intuition of how a tracing JIT compiler like RaptorJIT (and LuaJIT, its ancestor) works is that frequently executed code paths are detected and subsequently compiled. To detect what is hot and needs compiling the VM profiles the application at run time. The rationale being that paths that get executed often benefit from being optimized, while paths that are taken seldomly do not significantly affect total execution time, even if they are interpreted. Hey, compiling infrequently executed paths might not even be worth the overhead.

What LuaJIT, and by extension RaptorJIT, implements in reality goes like follows. The virtual machine maintains counters for branches in the program. Whenever the interpreter takes a branch a corresponding counter is incremented, and once that counter overflows a certain threshold the JIT attempts to compile a trace following that branch. At least, schematically it works like that.

So in effect, LuaJIT will attempt compile all code paths that hit the threshold eventually—first-come, first-served. That does not exactly match what was at least my intuition. Even branches taken rarely, say, once every minute or even every hour will eventually overflow their counters and trigger the JIT compiler.

One could argue that the system should not behave like that. The mere total number of executions of a given branch might not be a good metric to decide whether a code path is hot or not. Maybe there should be a connection to the time domain. Could we base compilation decisions on whether a code path has been executed a certain number of times within a given time period? Or as Luke Gorrie *commented* (https://github.com/lgalia/snabb/issues/1226#issuecomment-538967897): "Code gets hot when you hit it *frequently* and frequency is about both number of hits and length of time."

I hacked up a crude *patch* (https://github.com/raptorjit/raptorjit/pull/260) that resets the hot counters every second. Specifically, it checks if its time to reset the counters whenever a trace is to be compiled. If it turns out that the deadline has elapsed before the threshold overflow, it causes trace recording to bail and thus inhibits trace compilation. This way RaptorJIT compiles traces for branches that are taken at least N times within a second, instead of just N times overall. Initially, that patch was not without bugs. Andy Wingo helpfully pointed out that I overlooked the counters that track trace exits. Hence, `trace_clearsnapcounts` was born.

```
diff --git a/src/lj_trace.c b/src/lj_trace.c
index 5954771f..57184c1d 100644
--- a/src/lj_trace.c
+++ b/src/lj_trace.c
@@ -6,6 +6,8 @@
 #define lj_trace_c
 #define LUA_CORE

+#include <time.h>
+
 #include "lj_obj.h"


@@ -47,6 +49,45 @@ void lj_trace_err_info(jit_State *J, TraceError e)
   lj_err_throw(J->L, LUA_ERRRUN);
 }

+/* -- Hotcount decay ------------------------------------------------------ */
+
+/* We reset all hotcounts every second. This is a rough way to establish a
+** relation with elapsed time so that hotcounts provide a measure of frequency.
+**
+** The concrete goal is to ensure that the JIT will trace code that becomes hot
+** over a short duration, but not code that becomes hot over, say, the course
+** of an hour.
+**
+** The "one second" constant is certainly tunable.
+** */
+
+static void trace_clearsnapcounts(jit_State *J); /* Forward decl. */
+
+static inline uint64_t gettime_ns (void)
+{
+  struct timespec ts;
+  clock_gettime(CLOCK_MONOTONIC, &ts);
+  return ts.tv_sec * 1000000000LL + ts.tv_nsec;
+}
+
+/* Timestamp (ns) of last hotcount reset. */
+static uint64_t hotcount_decay_ts;
+
+/* Decay hotcounts every second. */
+int hotcount_decay (jit_State *J)
+{
+  uint64_t ts = gettime_ns();
+  int decay = (ts - hotcount_decay_ts) > 1000000000LL; /* 1s elapsed? */
+  if (decay) {
+    /* Reset hotcounts. */
+    lj_dispatch_init_hotcount(J2G(J));
```

The patch appears to work. At least, our immediate problems are solved, and as hypothesized the code paths that are no longer compiled after the patch do not significantly affect performance. I feel like this approach shows some promise.

Time	Description
00:00:32	trace stop (success) from lib/yang/alarms.lua:705 to (app)setvmprofile:1 (core/app.lua:95) (2875 bytecodes)
00:00:31	trace stop (success) from lib/yang/alarms.lua:705 to (app)setvmprofile:1 (core/app.lua:95) (71 bytecodes)
00:00:16	trace stop (success) from lib/yang/alarms.lua:328 to (lwaftr)LwAftr:push:29 (apps/lwaftr/lwaftr.lua:1100) (80 bytecodes)
00:00:12	trace stop (success) from lib/timers/ingress_drop_monitor.lua:76 to (app)with_restart:1 (core/app.lua:122) (2892 bytecodes)
00:00:10	trace stop (success) from lib/stream.lua:224 to (app)with_restart:1 (core/app.lua:122) (2631 bytecodes)
00:00:09	trace stop (success) from core/link.lua:88 to (echo)ICMPEcho:push:14 (apps/ipv6/echo.lua:152) (10 bytecodes)
00:00:09	trace stop (success) from core/link.lua:88 to (echo)ICMPEcho:push:14 (apps/ipv6/echo.lua:152) (10 bytecodes)
00:00:08	trace stop (success) from apps/lwaftr/lwaftr.lua:1117 to (lwaftr)LwAftr:flush_encapsulation:1 (apps/lwaftr/lwaftr.lua:763) (4 bytecodes)
00:00:08	trace stop (success) from apps/lwaftr/lwaftr.lua:1098 to (lwaftr)LwAftr:flush_encapsulation:1 (apps/lwaftr/lwaftr.lua:763) (50 bytecodes)
00:00:06	trace stop (success) from lib/ptree/alarms.lua:83 to (lwaftr)LwAftr:push:29 (apps/lwaftr/lwaftr.lua:1100) (1361 bytecodes)
00:00:06	trace stop (success) from apps/ipv6/echo.lua:155 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:05	trace stop (success) from lib/stream.lua:225 to (app)setvmprofile:1 (core/app.lua:95) (4085 bytecodes)
00:00:05	trace stop (success) from lib/stream.lua:223 to (stream)Stream:write_bytes:21 (lib/stream.lua:243) (207 bytecodes)
00:00:05	trace stop (success) from core/link.lua:88 to (echo)ICMPEcho:push:14 (apps/ipv6/echo.lua:152) (10 bytecodes)
00:00:05	trace stop (success) from apps/ipv6/echo.lua:155 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:05	trace stop (success) from apps/ipv6/echo.lua:155 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:05	trace stop (success) from apps/intel_mp/intel_mp.lua:757 to (app)setvmprofile:1 (core/app.lua:95) (48 bytecodes)
00:00:05	trace stop (success) from core/lib.lua:247 to (intel_mp)Intel82599:rxdrop:3 (apps/intel_mp/intel_mp.lua:1343) (339 bytecodes)
00:00:04	trace stop (success) from apps/intel_mp/intel_mp.lua:720 to (intel_mp)Intel:push:26 (apps/intel_mp/intel_mp.lua:720) (1 bytecodes)
00:00:04	trace stop (success) from core/link.lua:77 to (app)setvmprofile:1 (core/app.lua:95) (49 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/fragment.lua:331 to (app)setvmprofile:1 (core/app.lua:95) (28 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/fragment.lua:308 to (fragment)Fragmenter:push:24 (apps/ipv6/fragment.lua:308) (19 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/echo.lua:155 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:04	trace stop (success) from apps/lwaftr/ndp.lua:473 to (app)setvmprofile:1 (core/app.lua:95) (8 bytecodes)
00:00:04	trace stop (success) from apps/lwaftr/lwaftr.lua:1117 to (lwaftr)LwAftr:flush_encapsulation:1 (apps/lwaftr/lwaftr.lua:763) (4 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/echo.lua:159 to (app)setvmprofile:1 (core/app.lua:95) (30 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/arp.lua:260 to (app)setvmprofile:1 (core/app.lua:95) (29 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/reassemble.lua:326 to (app)setvmprofile:1 (core/app.lua:95) (18 bytecodes)
00:00:04	trace stop (success) from apps/intel_mp/intel_mp.lua:757 to (app)setvmprofile:1 (core/app.lua:95) (31 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/fragment.lua:308 to (fragment)Fragmenter:push:24 (apps/ipv6/fragment.lua:308) (19 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/echo.lua:155 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:04	trace stop (success) from apps/lwaftr/ndp.lua:473 to (app)setvmprofile:1 (core/app.lua:95) (8 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/echo.lua:159 to (app)setvmprofile:1 (core/app.lua:95) (30 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/arp.lua:260 to (app)setvmprofile:1 (core/app.lua:95) (29 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/reassemble.lua:326 to (app)setvmprofile:1 (core/app.lua:95) (18 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (echo)ICMPEcho:push:14 (apps/ipv6/echo.lua:152) (10 bytecodes)
00:00:04	trace stop (success) from apps/intel_mp/intel_mp.lua:727 to (app)setvmprofile:1 (core/app.lua:95) (28 bytecodes)
00:00:04	trace stop (success) from apps/lwaftr/lwaftr.lua:1117 to (lwaftr)LwAftr:flush_encapsulation:1 (apps/lwaftr/lwaftr.lua:763) (4 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (fragment)Fragmenter:push:24 (apps/ipv6/fragment.lua:308) (10 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (ndp)NDP:push:23 (apps/lwaftr/ndp.lua:467) (10 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (lwaftr)LwAftr:push:29 (apps/lwaftr/lwaftr.lua:1100) (10 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (echo)ICMPEcho:push:3 (apps/ipv4/echo.lua:149) (10 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (arp)ARP:push:7 (apps/ipv4/arp.lua:230) (10 bytecodes)
00:00:04	trace stop (success) from core/link.lua:88 to (reassemble)Reassembler:push:6 (apps/ipv4/reassemble.lua:301) (10 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/fragment.lua:192 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/echo.lua:163 to (app)setvmprofile:1 (core/app.lua:95) (7 bytecodes)
00:00:04	trace stop (success) from apps/ipv4/arp.lua:267 to (app)setvmprofile:1 (core/app.lua:95) (8 bytecodes)
00:00:04	trace stop (success) from apps/lwaftr/lwaftr.lua:1098 to (binding_table)BTLookupQueue:process_queue:4 (apps/lwaftr/binding_table.lua:106) (18 bytecodes)
00:00:04	trace stop (success) from apps/ipv6/echo.lua:151 to (echo)ICMPEcho:push:14 (apps/ipv6/echo.lua:152) (23 bytecodes)

Trace creation timeline after the patch is applied. Most traces are recorded and compiled within seconds of starting the benchmark.