

Lightweight Preempt and IRQ Tracepoints for Production RT Kernels

Wander Lairson Costa

wander@redhat.com

<https://walac.github.io>

01. THE TRIGGER

Latency Case Spikes

Customer reports unpredictable latency spikes in production.

Finding: Long period of preemption or disabled IRQ.

02. THE OBSTACLES

Replication Barriers

No Lab Repro: Cannot reproduce the latency issues in a lab environment.

No Custom Kernel: Deploying custom test kernels in production is infeasible.

03. THE OBJECTIVE

Pinpoint Source

We need to discover precisely where the latency originates.

Requires safe, low-overhead tracing in live client systems.

Inline check if the tracepoint static-key

CONFIDENTIAL designator

```
void __no_caller_saved_registers
trace_local_irq_enable(void);
#define local_irq_enable() \
do { \
    if (tracepoint_enabled(irq_enable)) \
        __trace_call(trace_local_irq_enable); \
    raw_local_irq_enable(); \
} while (0)
```

```
    nop
1: call _arch_local_irq_enable
...
2: call _trace_local_irq_enable
    jmp 1b
```

```
/* arch/x86/include/asm/trace.h */  
  
void __no_caller_saved_registers  
trace_local_irq_enable(void);  
  
if (tracepoint_enabled(irq_enable))  
    asm volatile("call trace_local_irq_enable"  
                : : "memory");
```

FAST-PATH SPILL RISK

Standard compiler behavior forces register spills before evaluating branch checks.

REGISTER PRESERVATION

Custom infrastructure utilizes `__no_caller_saved_registers` to avoid spilling (gcc implements for x86).

FAST-PATH OPTIMIZATION

Uses inline assembly to hide the call from standard compiler register allocation (gcc limitation).

Static binary overhead analysis

CONFIDENTIAL designator

Analysis Metric	Value
Functions analyzed	1,125
• Skipped (not in both builds)	67
• Flagged (inlining diffs)	18
Total difference	+9,752 instructions (+7.25%)
Total trace call sites	3,080
Avg overhead/call site	3.2 insns

Overhead Distribution

Instructions per call site



iperf3 – benchmark comparison – kernel-rt 7.2.0

CONFIDENTIAL

INTEL(R) XEON(R) GOLD 6542Y - 48 cores - 96 CPUs - L1d=2MB L1i=1.5MB L2=96MB L3=120MB

not significant, (*) = $p < 0.05$, (**) = $p < 0.01$

Metric	no-preemptirq	preemptirq	IRQSOFF/PREEMPT
Throughput Metrics (More is better)			
tcp_receiver_gbps	73.72 Gbps	-0.0% (ns)	-5.1% (**)
tcp_sender_gbps	73.66 Gbps	+0.2% (**)	-5.1% (**)
udp_receiver_gbps	51.02 Gbps	-1.0% (**)	-5.2% (**)
udp_sender_gbps	51.76 Gbps	-0.4% (**)	-4.2% (**)
Hardware Metrics & Cache Events (Less is better)			
perf:L1-dcache-load-misses	2,664,085,150	-0.5%	-5.2%
perf:L1-icache-load-misses	2,740,947,889	-2.0%	-2.0%
perf:LLC-load-misses	277,077	+19.5%	+45.9%
perf:branch-misses	47,644,593	-0.6%	+7.2%
perf:cache-misses	1,334,674	+5.8%	+8.1%
perf:dTLB-load-misses	245,851	-3.5%	-54.3%
perf:iTLB-load-misses	31,914,390	-0.8%	-1.0%

iperf3 – benchmark comparison – kernel-rt 7.2.0

AMD EPYC 7252 16 cores L1=512KB L2=8MB L3=128MB

not significant, (*) = $p < 0.05$, (**) = $p < 0.01$

Metric	no-preemptirq	preemptirq	tracer
Throughput Metrics (More is better)			
tcp_receiver_gbps	21.27 Gbps	-0.7% (ns)	-3.6% (**)
tcp_sender_gbps	18.89 Gbps	+1.2% (ns)	+1.4% (*)
udp_receiver_gbps	50.32 Gbps	-0.9% (**)	-12.6% (**)
udp_sender_gbps	53.77 Gbps	-0.6% (**)	-11.3% (**)
Hardware Metrics & Cache Events (Less is better)			
perf:L1-dcache-load-misses	1,477,840,479	+5.8%	+13.5%
perf:L1-icache-load-misses	81,108,120	+18.0%	+29.3%
perf:branch-misses	21,000,230	+11.6%	+14.8%
perf:cache-misses	433,274,469	+15.0%	+14.6%
perf:dTLB-load-misses	831,544	+60.1%	+26.8%
perf:iTLB-load-misses	21,584	+282.4%	+88.2%

Cold Code Isolation

```
void function() {  
    if (unlikely(key))  
        goto cold_path;  
    // Main execution path continues...  
    ...  
    __attribute__((section(".text.unlikely")))  
    cold_path: /* isolated */  
}
```

Proposed Optimization

Move generated cold code to dedicated sections like `.text.unlikely`.

01. Better Isolation

Isolates call setup and argument setup completely.

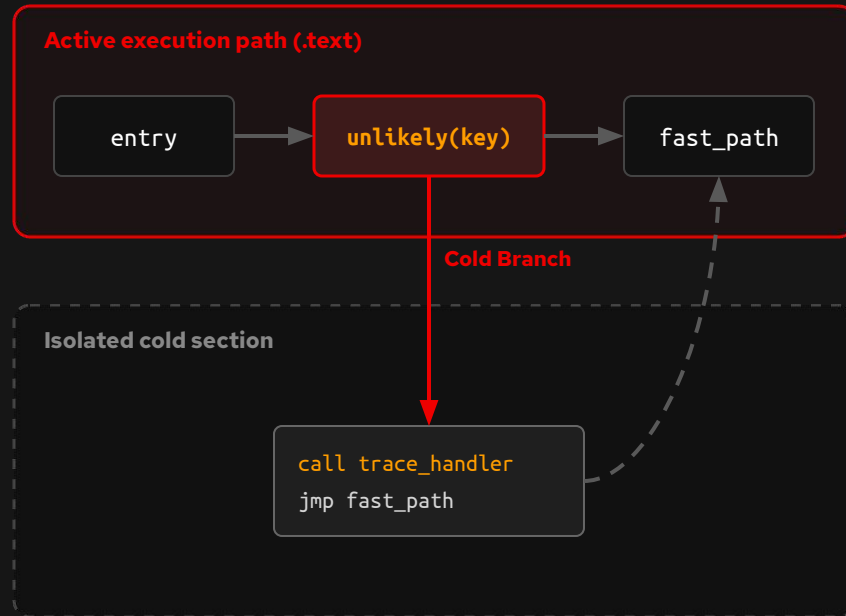
02. Prevent Interleaving

Prevents cold code from interleaving with the main execution path.

03. Optimize Cache

Significantly reduces instruction cache pressure and cache line pollution.

Instruction Stream Organization



Static Branch Call Optimization



```
// Traditional: Call & setup pollute the cache  
if (static_branch_unlikely(&key))  
    handler(arg);
```

```
// Proposed: "Zero" inline call footprint
```

```
static_branch_call_unlikely(&key, handler);
```

```
// to handle local_irq_save/restore(flags)
```

```
static_branch_call_arg1_unlikely(&key, handler, arg);
```

01. Zero Fast-Path Overhead

Emits only the inline jump-label NOP/JMP. Call instructions, argument register loading, and stack frame adjustments are completely removed from the cache when the key is disabled.

02. Out-of-Line Section Routing

Directs execution to dedicated `.ref.text.jl.handler` sections, preventing cold code from interleaving with active instruction cache lines.

03. Register & ABI Safety

Out-of-line trampolines save and restore caller-saved registers (bypassed if `__no_caller_saved_registers` is present).

Thank you.

Questions & Discussion

Wander Costa
wander@redhat.com