Extending the *extended* BPF

KeBPF $\leftrightarrow$ UeBPF

Suchakrapani Datt Sharma

May 13, 2015

École Polytechnique de Montréal

Laboratoire **DORSAL**

Agenda

Recap

- Research Updates

Investigations

- Background
- An experimental userspace eBPF library
- Performance of Userspace eBPF and LTTng filters
- Extensions to Kernel eBPF, example use-case

Upcoming and in-progress

- Explore KeBPF ↔ UeBPF interactions

Recap

Research Focus : Integrated and streamlined framework for tracing & debugging, dynamic instrumentation & JIT techniques

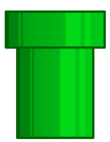
focus, more focus

- Explore more of eBPF + Tracing
- Rapid developments on kernel side mean more opportunities
- Extensions of eBPF in assisted-tracing

Recap

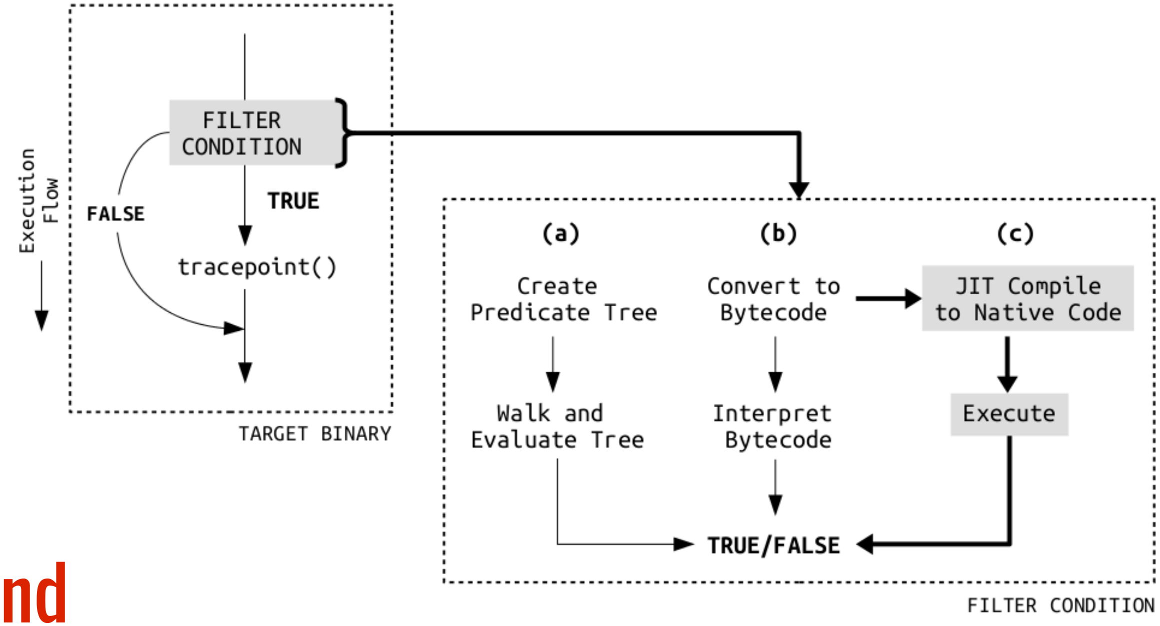
Where we left off

- Evaluating pure performance of eBPF+JIT in kernel
- Observing it's performance with LTTng kernel tracing
    - Interpreted eBPF - 83ns/event, JITed eBPF - 25ns/event with a simple filter

In the  last time

- Userspace eBPF for better control and comparisons with LTTng
- Exploit opportunities to improve eBPF JIT internals
    - Recent developments in LLVM backend for eBPF [1]
- Assisted tracing, explore actions in eBPF

Investigations



Background

- Extended Berkeley Packet Filter (**eBPF**)
  - Fast, small, in-kernel packet & syscall filtering [2]
  - Register based, switch-dispatch interpreter
  - Special BPF syscall, 64-bit regs, shared-map operations

Investigations

Why eBPF in Tracing

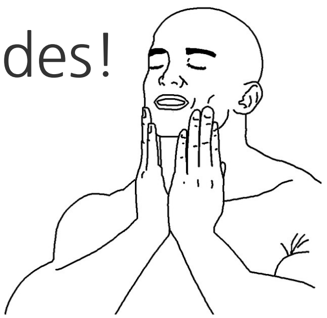
- Primarily for filters & script driven tracing
- Add sophisticated features to tracing, at low cost
  - Fast stateful kernel event filtering
    - In trace-synchronizarion to reduce overhead by only selecting specific packets matching criteria
    - Record system wide sched_wakeup only when target process is blocked to reduce overhead
  - Utilize *side-effects* for assisted-tracing (exploit fall-through)
- A more uniform way of filtering events across userspace and kernel

Sample use-cases

Investigations

Userspace eBPF (UeBPF)

- Experimental *libebpf* to provide filtering in userspace tracing
- Includes side-effects through communication with modified KeBPF
- Easy switch between JIT/interpret for performance analysis
- Includes LLVM backend [1] No more raw bytecodes!
- Load bytecode from eBPF binaries

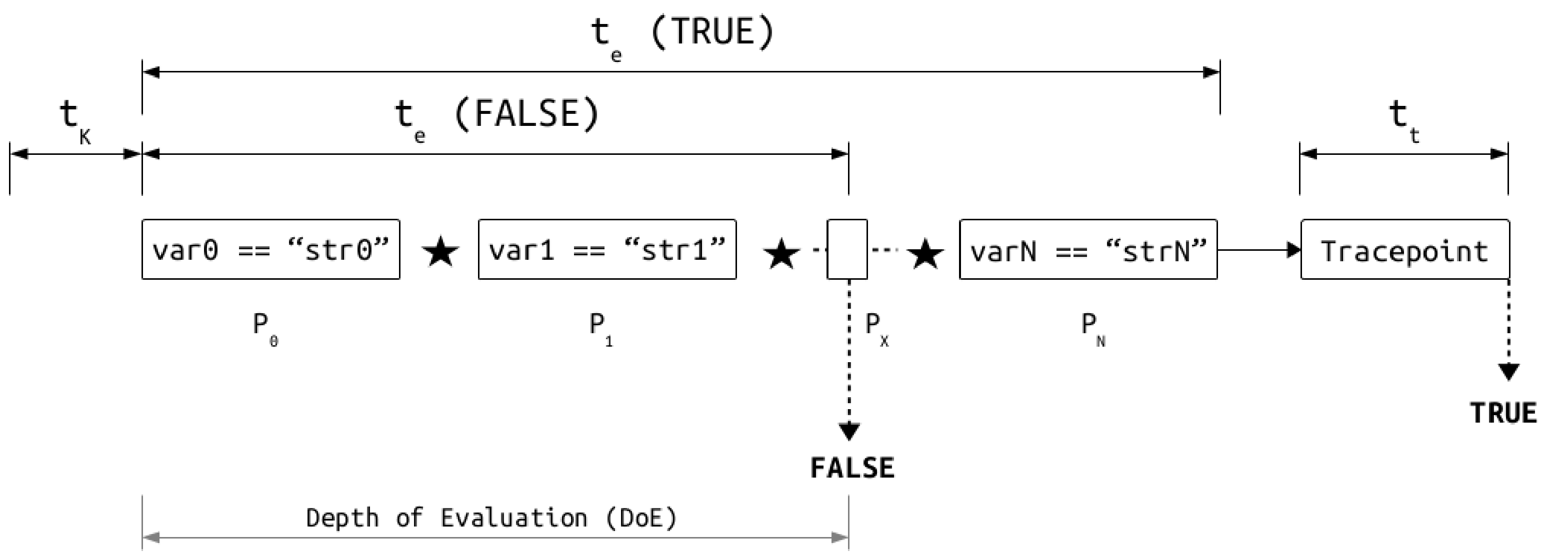


Performance Analysis

- Apply LTTng, eBPF, eBPF+JIT, hardcoded filters
- Measure $t_{execution} + t_{tracepoint}$

Investigations

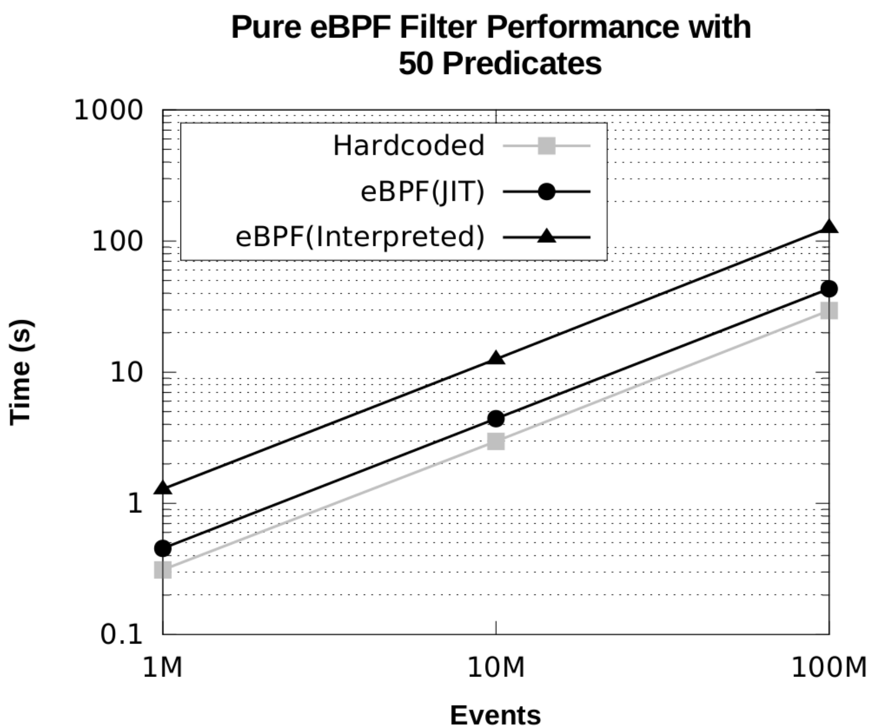
Performance Analysis

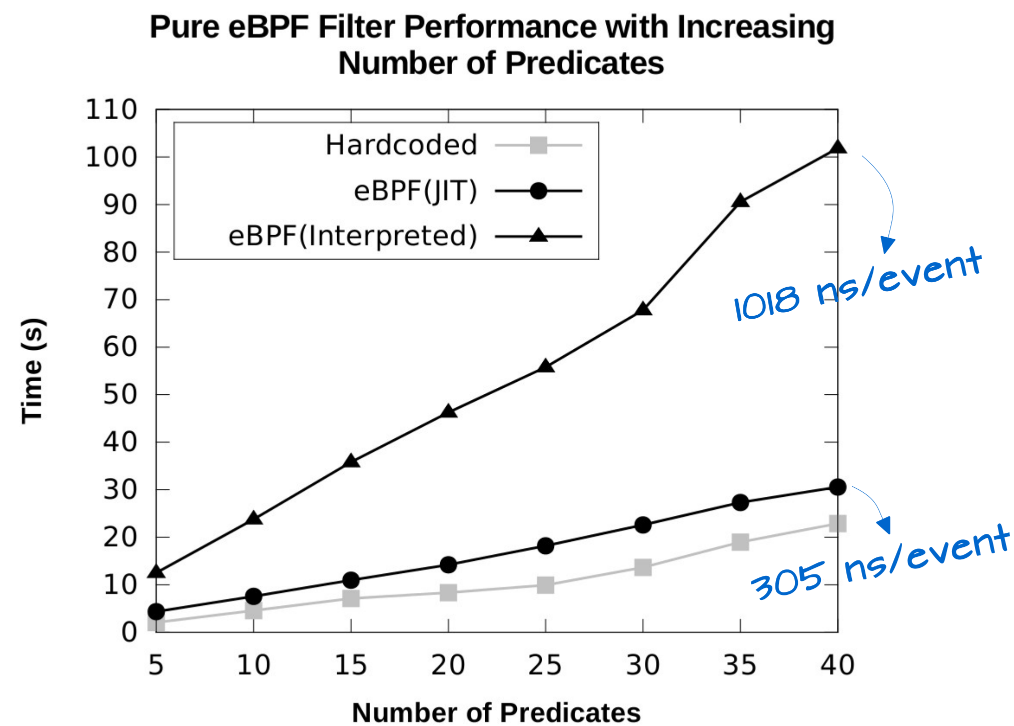


- Pure filter evaluation.
  - TRUE/FALSE biased AND chain with varying predicates
- Measure $t_e + t_t$ with varying *DoE* (Biased TRUE)

Investigations

Performance Analysis

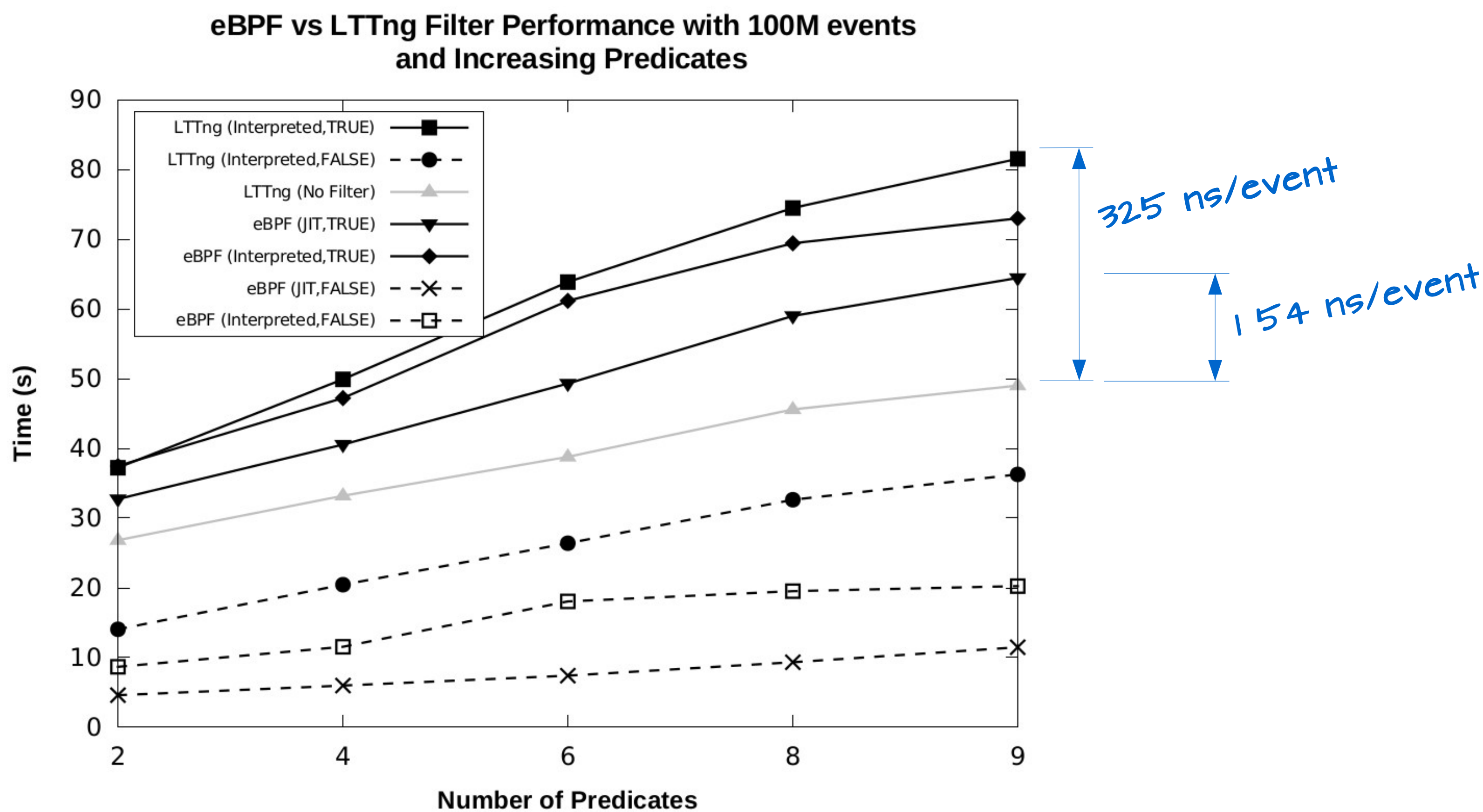




- Steady gain in **3x** range for JIT vs Interpreted with increasing events, slightly increasing gain (**3.1x** to **3.3x**) with increasing predicates

Investigations

Performance Analysis



- eBPF JIT*ed* filter is **3.1x** faster than LTTng's interpreted bytecode and eBPF's interpreted filter is **1.8x** faster than LTTng's interpreted version

In-progress

KeBPF ↔ UeBPF Extensions

- Syscall latency tracking use-case. Thank you François, Francis and Julien
- Latency threshold is defined statically and manually [3]
  - In real life, it may need to be set dynamically - different machines can have different *normal levels* for syscalls
  - We may need to adaptively set thresholds per syscall based on user's criteria as well as tracking the *normal* behaviour.
  - We can use eBPF *side-effects* to provide dynamic and adaptive thresholds

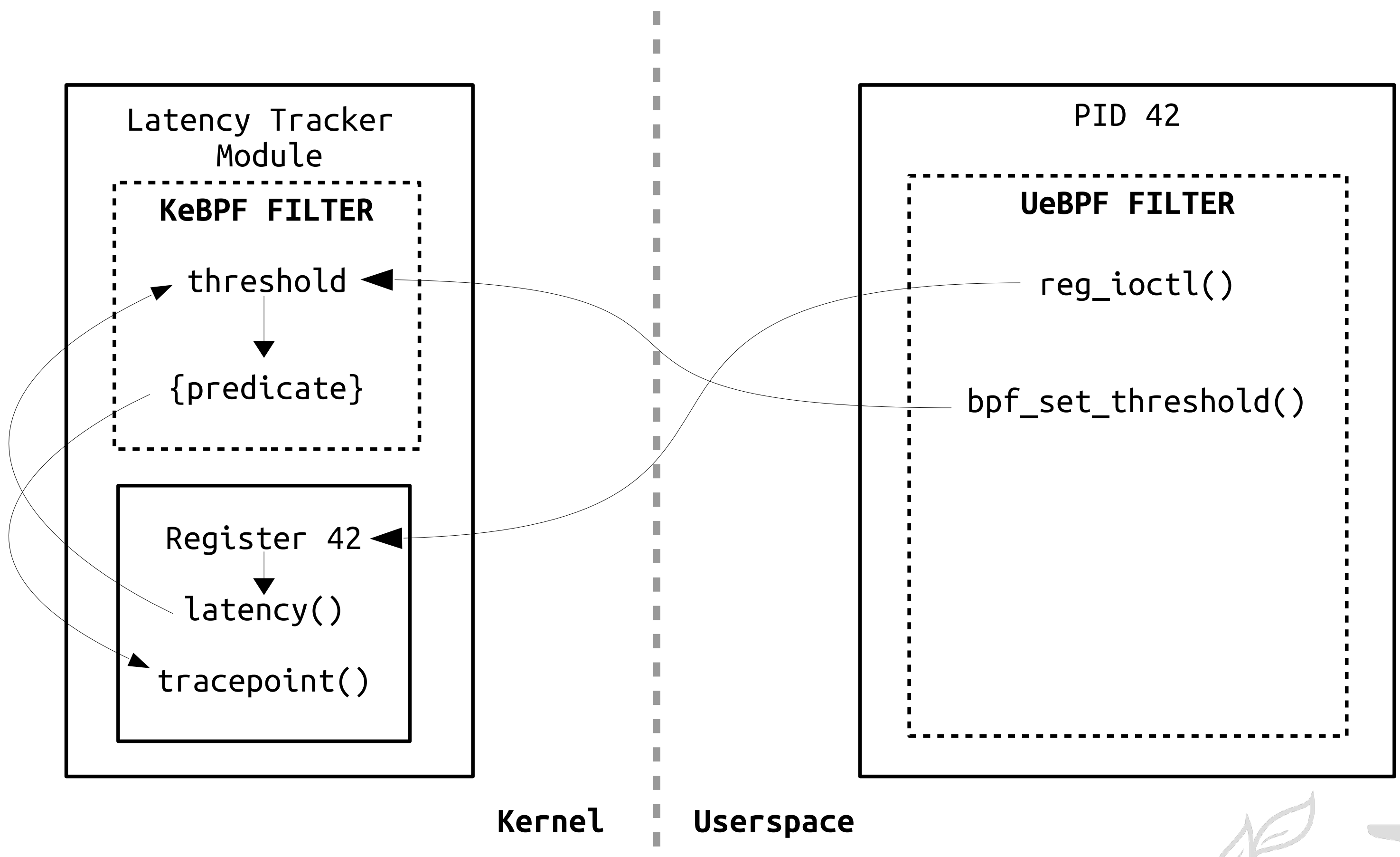
In-progress

KeBPF ↔ UeBPF Extensions

- Side-effects?
  - eBPF is not just JT/JF targets [6], we can do more complex things like perform internal actions in addition to decisions
  - We can implement more internal BPF helper functions such as bpf_get_threshold(), bpf_prof_analysis() etc.
  - Access shared data from KeBPF/UeBPF
  - Maintain such states within eBPF and use side-effects to compute complex decisions

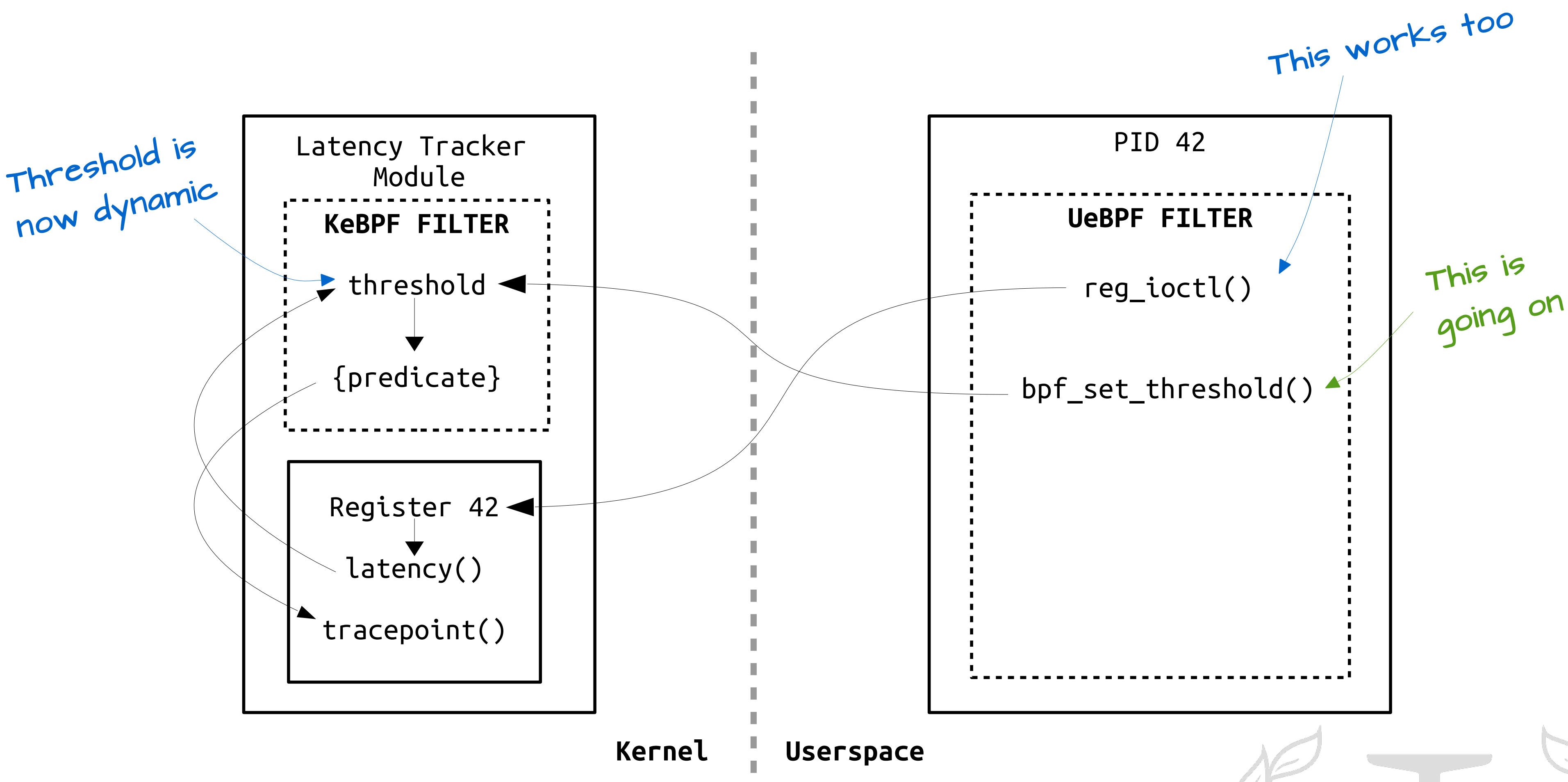
In-progress

KeBPF ↔ UeBPF Syscall Latency Tracking



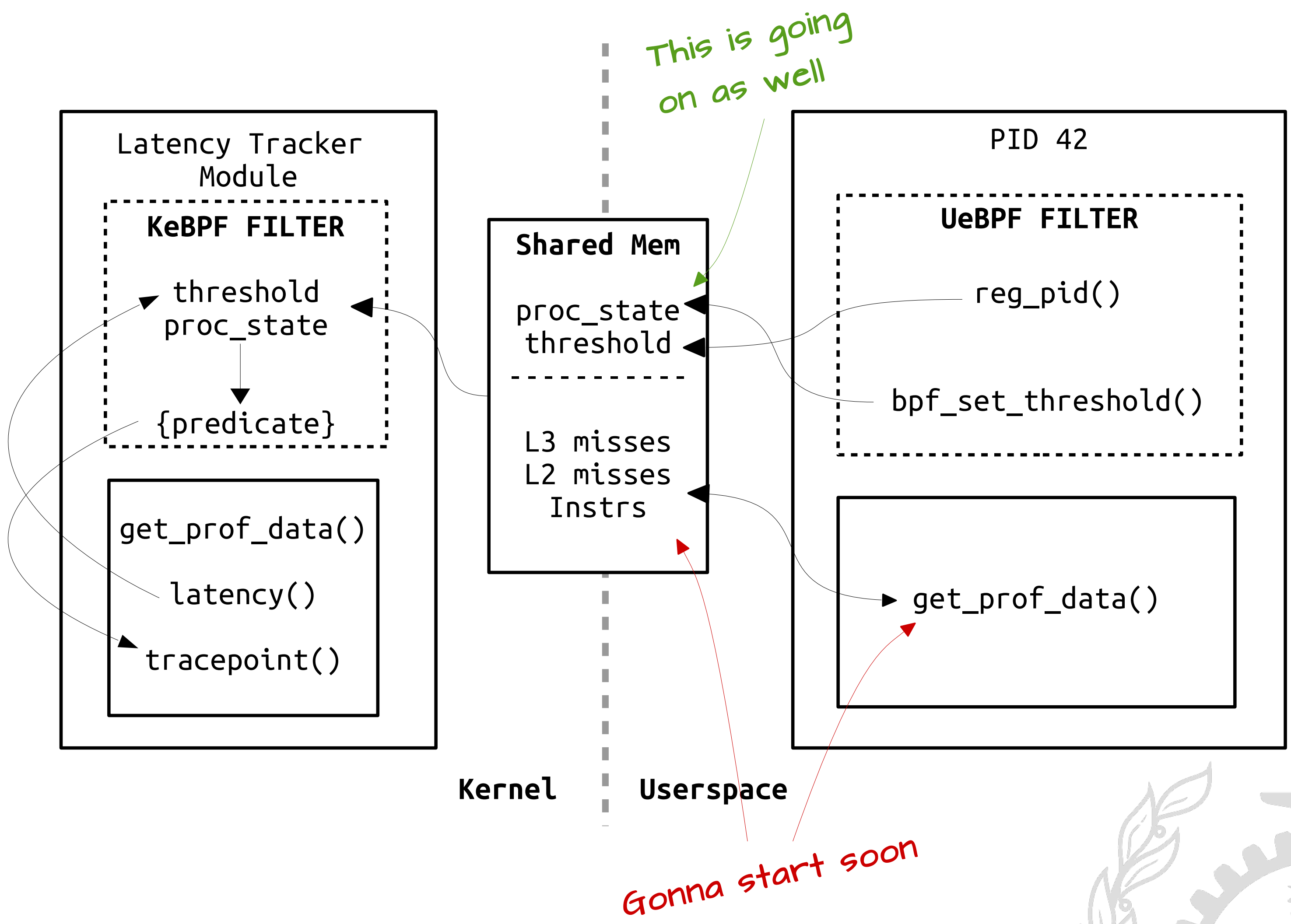
In-progress

KeBPF ↔ UeBPF Syscall Latency Tracking



In-progress

KeBPF ↔ UeBPF Syscall Latency Tracking



In-progress

KeBPF ↔ UeBPF Syscall Latency Tracking

- Shared Memory
  - Mix of RCU based hash table and atomic value array
  - Perf-like implementation mmap+debugfs probably
  - L2, L3 cache misses, instructions retired per CPU and other profiling data can be reliably obtained from special instructions using Perf/PAPI [4]
  - More data (reads/writes to memory controller) on specific Intel archs using other direct ring-1 mode privileged instructions or MSR module [5]

What's Next

Inferences

- UeBPF is fast, a good port for performance comparison
- We can use it further to develop userspace assisted kernel tracing or kernel assisted userspace tracing
  - Such as, recording kernel events at function-granularity

Going Further

- Decide upon scalable kernel-user data sharing approach
- New helper functions from within eBPF to handle this data and explore more side-effects with states (eg. synchronization)
- Consider ABI-less mechanism of transferring data

References

[1] http://reviews.llvm.org/rL227008

[2] https://www.kernel.org/doc/Documentation/networking/filter.txt

[3] https://github.com/fdoray/lttng-profile

[4] Terpstra, D., Jagode, H., You, H., Dongarra, J. "Collecting Performance Data with PAPI-C," Tools for High Performance Computing 2009, Springer Berlin / Heidelberg, 3rd Parallel Tools Workshop, Dresden, Germany, pp. 157-173, 2009

[5] https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[6] Schulist, J., Borkmann, D., Starovoitov, A.: Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt

Questions?

suchakrapani.sharma@polymtl.ca

suchakra on #lttng