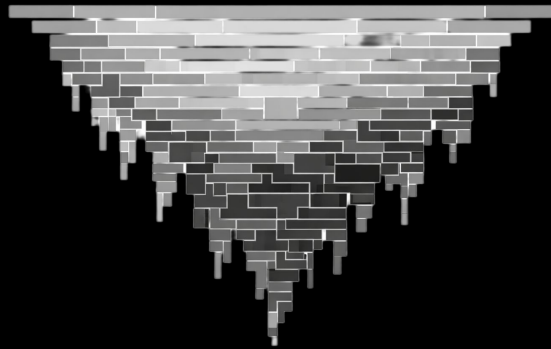


Profiling



Who

Noam Levy

Field CTO

groundcover



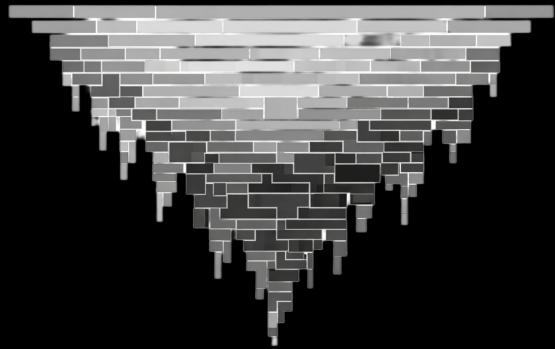
What

Profiling is
powerful

It's (Becoming)
accessible

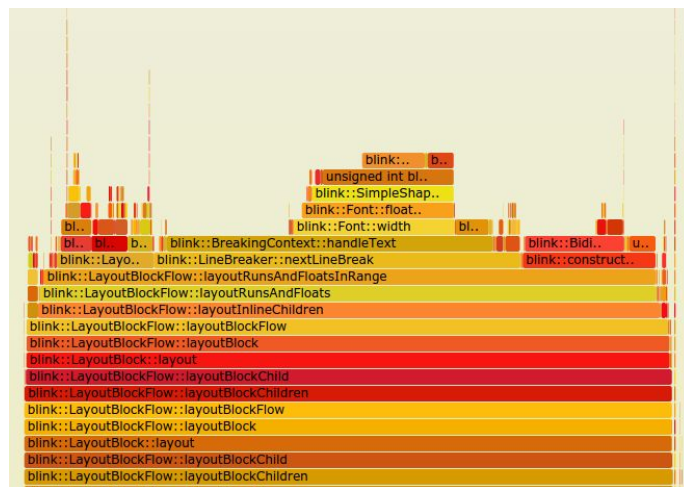
And soon enough,
~frictionless

Powerful



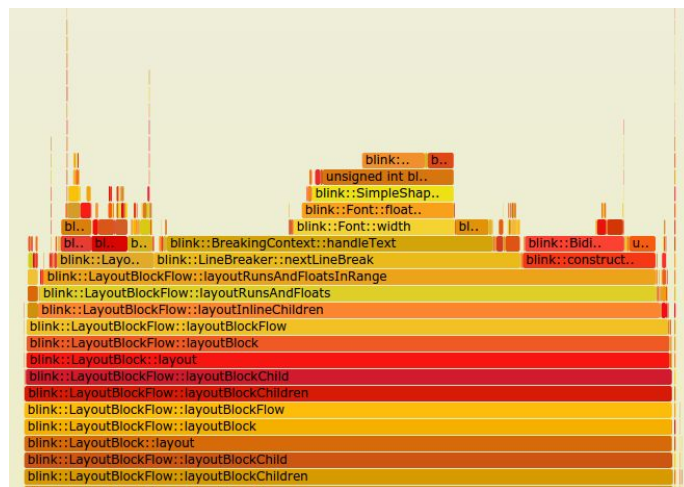
101

- Resources Allocation Behavior Tracking
- CPU or Memory
- Continuous/Ad Hoc
- Down to the function

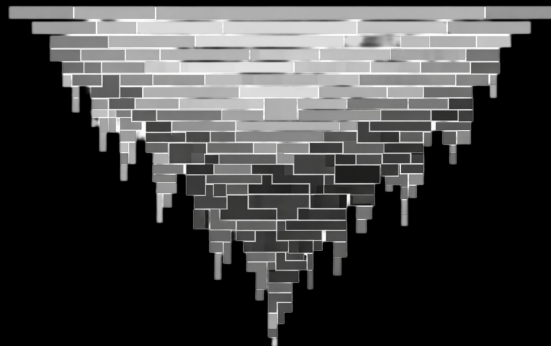


Words

- Flame graphs
- Winding/Walking
- Symbolization
- **eBPF**



Accessible



Accessibility

- How easy it is to **implement**?
- How easy it is to **consume**?

Implement

- Profiling was a fragmented field
- Runtime sensitive
- Usually operated separately
- Binaries/OS were not always compatible
- **Isolated pillar**



Implement

- Framework for standardizing telemetry delivery
- Spec and Collector
- Metrics, Logs, Traces
- **Profiling signal released**



Implement

- Portable / Vendor neutral
- Popular, **referable**
- Multiple runtimes
- Consistent



Implement

- Receiver
- Processor
- Exporter

```
receivers:  
  otlp:  
    protocols:  
      grpc:  
        endpoint: 0.0.0.0:4317  
  
processors:  
  batch: {}  
  
exporters:  
  otlp/backend:  
    endpoint: <>  
  
service:  
  pipelines:  
    traces:  
      receivers: [otlp]  
      processors: [batch]  
      exporters: [otlp/backend]
```

Implement

- SDK (instrumentation)
- *not always needed

```
func (c *clickHouseHttpClient) Do(ctx context.Context, req *http.Request) (*http.Response, []byte, error) {  
    ctx, span := serverTracer.Start(ctx, clickhouseServerDoSpan)  
    defer span.End()
```

Consume

- Profiles also contain information about **the logic peripherals**
- Its is hard to understand
- Only few (**humans**) could make decisions based of it.

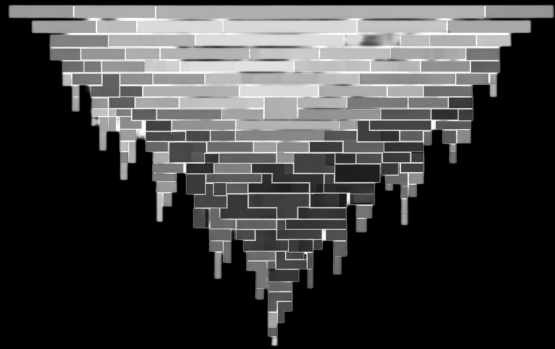


Consume

- If Agents can read profiles
- Agents can take you straight to conclusions
- **The bar for benefiting from profiling has been lowered**



Friction(less?)



eBPF

- Kernel sandbox for loading programs with **access to the entire os and running apps**
- Allows to **hook on** kernel and user programs at entry/exit/custom places.
- *"This is like putting Javascript into the kernel."* - Brendan Gregg



eBPF 101

- We hook
- We work (safely)
- We leave

```
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    pid_t pid;
    const char *filename;

    pid = bpf_get_current_pid_tgid() >> 32;
    filename = BPF_CORE_READ(name, name);
    bpf_printk("KPROBE ENTRY pid = %d, filename = %s\n", pid, filename);
    return 0;
}

SEC("kretprobe/do_unlinkat")
int BPF_KRETPROBE(do_unlinkat_exit, long ret)
{
    pid_t pid;

    pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("KPROBE EXIT: pid = %d, ret = %ld\n", pid, ret);
    return 0;
}
```

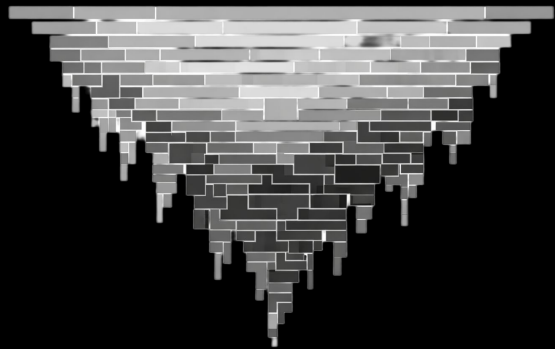
eBPF+O11Y

- eBPF unblocks **zero instrumentation techniques**
- We can automate signal generation **without relying on engineers**
- Used for **generating** traces, metrics
- And now - **profiling**

eBPF+Profiling

- Opentelemetry-ebpf-profiler
- Wide support, including (kinda) stripped binaries
- **No need for instrumentation**

Getting Started



Getting started

- eBPF agent
- OpenTelemetry Collector
- Profiling backend
- *K8s



Profiling Agent

- eBPF agent is **privileged**
- hostPID: true
- securityContext.privileged = true
- /sys, /proc, /lib volume mounts

```
spec:  
  hostPID: true  
  containers:  
    - name: profiler  
      image: otel/opentelemetry-collector-ebpf-profiler:latest  
      args: --  
      env: --  
      resources: --  
      securityContext:  
        runAsUser: 0  
        runAsGroup: 0  
        privileged: true  
      volumeMounts:  
        - name: config  
          mountPath: /etc/otel/config.yaml  
          subPath: config.yaml  
        - name: sys-kernel  
          mountPath: /sys/kernel  
          readOnly: true  
        - name: tracefs  
          mountPath: /sys/kernel/tracing  
          readOnly: true  
        - name: lib-modules  
          mountPath: /lib/modules  
          readOnly: true  
        - name: proc  
          mountPath: /proc  
          readOnly: true
```

Collector

- Receiver
 - **OTLP**
- Processors
 - **Metadata enriching**
- Exporter
 - **Profiling backend**

```
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317

processors:
  k8sattributes:
    auth_type: "serviceAccount"
    passthrough: false
    extract: --
    pod_association:
      - sources:
          - from: resource_attribute
            name: container.id

exporters:
  otlp_http/pyroscope:
    endpoint: http://pyroscope.profiling-lab.svc.cluster.local:4040

service:
  pipelines:
    profiles:
      receivers: [otlp]
      processors: [k8sattributes]
      exporters: [otlp_http/pyroscope]
```

Backend

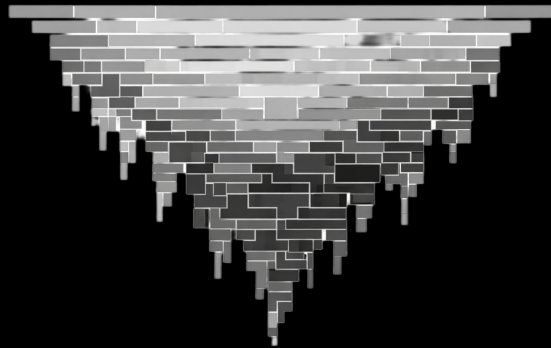
- Needs storage
- StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: pyroscope
  namespace: profiling-lab
spec:
  serviceName: pyroscope
  replicas: 1
  selector: ...
  template:
    metadata: ...
    spec:
      containers:
        - name: pyroscope
          image: grafana/pyroscope:latest
          args: ...
          ports: ...
          volumeMounts: ...
      volumeClaimTemplates:
```

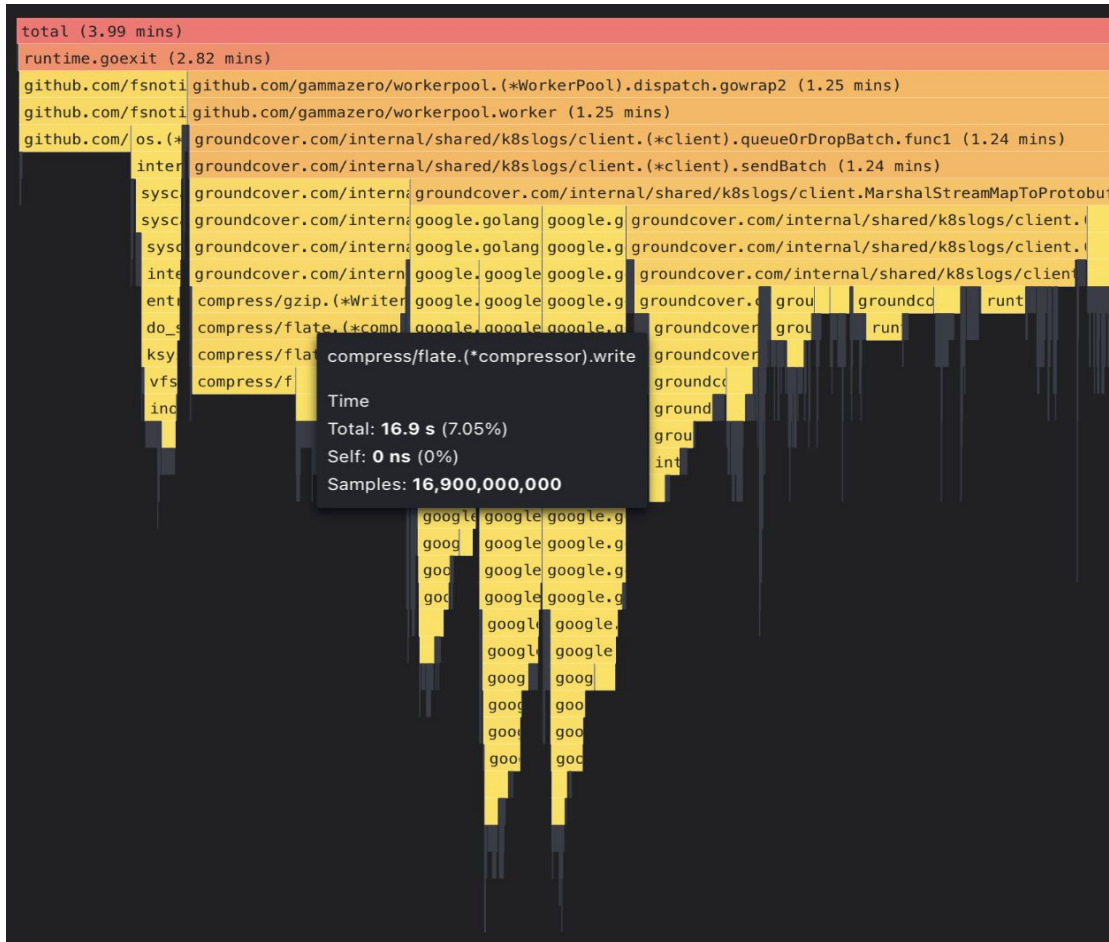

Not Perfect

- Still early
- Symbolization
- Making sense out of it

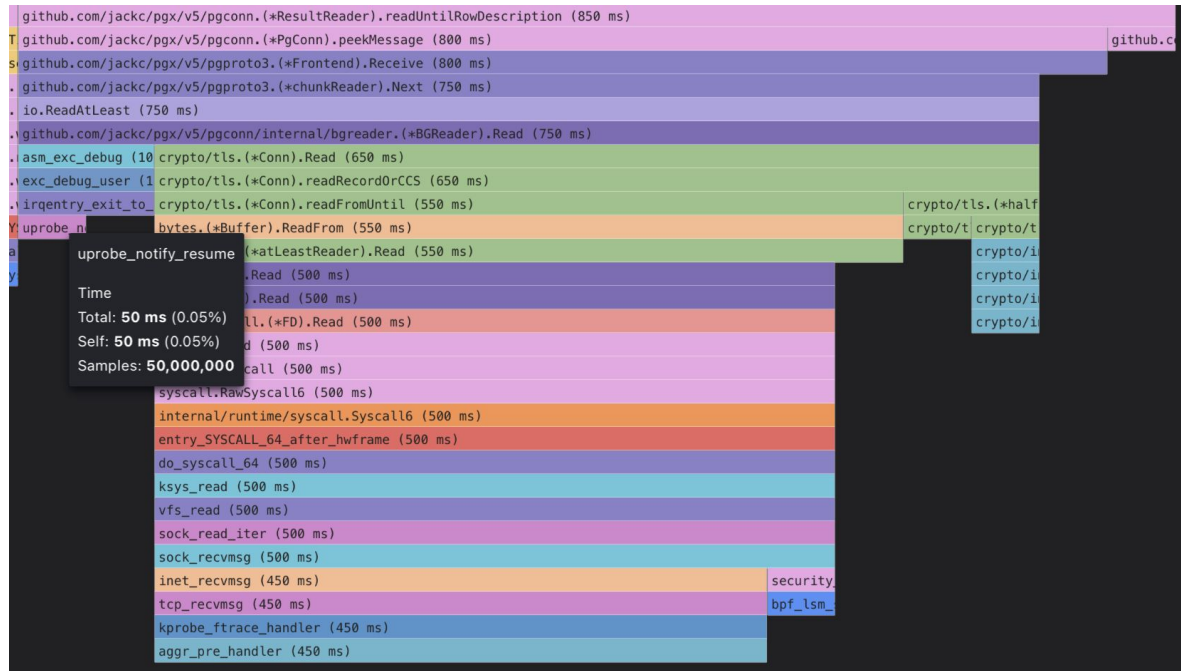
Examples



Time spent on serialization



eBPF overhead!



Much more

- 3rd party effects
- OS
- Memory profiling

The 4th Pillar

- Profiling insights **complements MLT**
- Latency from metric - drill down to a span - bottleneck from profile
- Suspicion from a profile - added instrumentation - clearer signal going forward
- **Single pane of glass**

AI

Single | process_cpu

http://localhost:4040

router:process_cpu:cpu • Last 30 minutes

Select Tag process_cpu:cpu:nanoseconds:cpu:nanoseconds(service_name="router", service_namespace="groundcover-main") Execute

Total nanoseconds of CPU time consumed

Search...

1.85 mins | 111 Bil samples (Time)

total (1.85 mins)

Category	Sub-category	Value
runtime	runtime.goexit	(51.1 s)
runtime	runtime.systemstack	(51.4 s)
runtime	runtime.gcBgMarkWorker.func2	(49.1 s)
runtime	runtime.gcDrain	(48.9 s)
runtime	runtime.scanobject	(28 s)
runtime	runtime.getempt	
runtime	runtime.handoff	
runtime	runtime.greypol	
runtime	runtime.gcDrain	9.45s self / 48.9s total
runtime	runtime.greypol	10.4s self
runtime	runtime.scanobject	6.3s self
runtime	runtime.gcDrain	9.45s self / 48.9s total
runtime	runtime.greypol	10.4s self
runtime	runtime.scanobject	6.3s self
runtime	runtime.gcDrain	9.45s self / 48.9s total
runtime	runtime.greypol	10.4s self
runtime	runtime.scanobject	6.3s self

Yes.

What stands out is not "business logic is hot," but "the Go runtime and allocation/GC path are dominating."

Top signals from the profile:

- Allocation pressure is very high**
 - `aggr_pre_handler` is the top self-time frame at **15.5s self / 16.2s total**
 - right below it: `runtime.getempt`, `runtime.greyobject`, `runtime.gcDrain`, `runtime.scanobject`, `runtime.trygetfull`, `runtime.findObject`
 - that pattern usually means the handler is creating a lot of short-lived objects, and the runtime is paying for allocation + mark/scan work.
- GC is a major bottleneck**
 - `runtime.gcDrain` shows **9.45s self / 48.9s total**
 - `runtime.gcDrain` **10.4s self**
 - `runtime.greypol` **6.3s self**
 - this is classic "CPU burned in garbage collection because object churn is high."
- JSON decoding is non-trivial and likely part of the churn source**
 - `encoding/json.(*Decoder).readValue`
 - `encoding/json.stateInString`
 - `encoding/json.(*decodeState).skip`
 - `encoding/json.(*decodeState).rescanLiteral`
 - `encoding/json.unquoteBytes`

These are not the absolute top cost, but they are visible enough to matter, and `encoding/json` often causes allocation-heavy behavior if decoding into generic maps, interfaces, or repeatedly building temporary structs/strings.
- The flame graph suggests the hot path is request-serving related**
 - visible stacks include `net/http.(*Server).Serve`, `net/http.serverHandler.ServeHTTP`, `github.com/gin-gonic/gin...`
 - also OpenTelemetry/gRPC-related frames appear in the stack
 - that implies the profile is from active request handling, not background-only work.

My read:

- The primary issue is probably **memory churn inside `aggr_pre_handler`**, not raw compute.

Single | process_cr

Ask anything

AI+eBPF

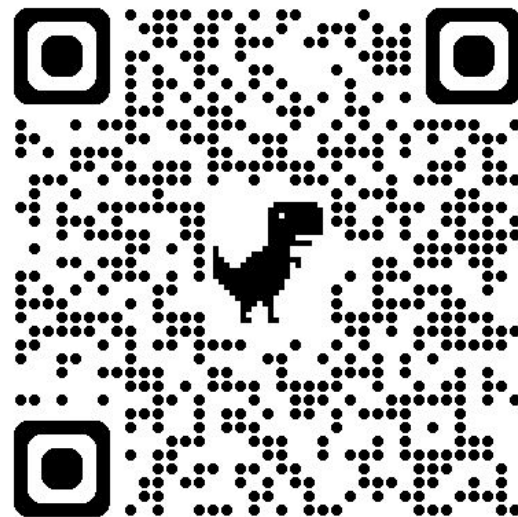
- AI will analyze profiles **without/alongside humans**
- eBPF **is** generating signals without humans
- They empower each other
- Unblocks human-on-the-loop

Takeaways

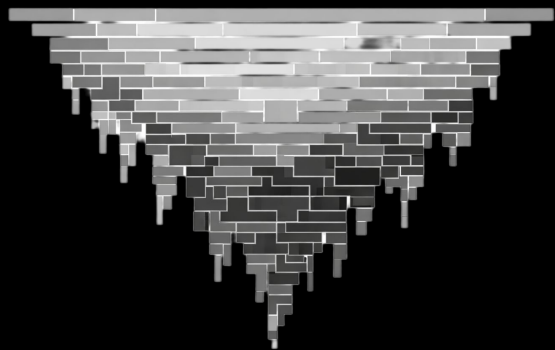
- Introducing profiling becomes **easier**
- tooling/approach is **standardized**
- It's only getting **better**
- There is so much your apps are doing **and you don't know**
- eBPF unblocks human-agnostic observability.
- <https://www.brendangregg.com/>

Try it yourself

- One command
- Full setup -
agent,collector,backend



?



groundcover

groundcover

- eBPF based O11Y platform
- OpenTelemetry native
- BYOC Architecture
- AI packed
- Profiling - soon

