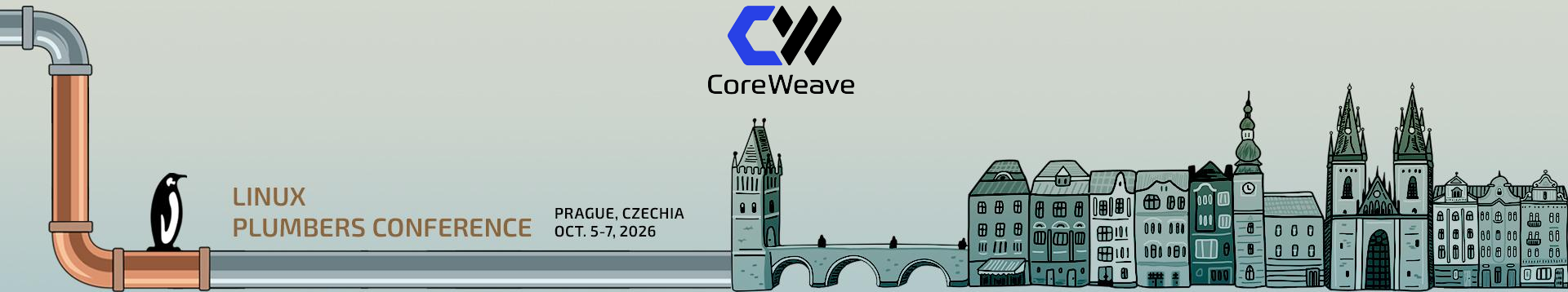


Crash Dumps In A Bare-Metal Cloud Provider

Serapheim Dimitropoulos



Context & Assumptions

- Customers rent whole physical rack/nodes
 - Can't easily move them away during failures - quick recovery is important
- Majority of nodes are expensive machines that are in constant demand
 - Can't easily recreate customer setup and repro issues - crash dumps are essential
- Nodes are pretty big (2-4TBs of RAM)
 - Default/Single-threaded makedumpfile can take anywhere from 20mins to an 1hr.
- Wide variety of failure modes - HW, FW, cables, etc..
 - Need a way to distinguish actual kernel issues from the rest of the issues
 - Also distinguish between known issues vs new issues

What We Do Currently

- A stripped crash kernel after a panic (4 CPUs enabled)
- Mounts fs with local DB of known panic fingerprints
- Checks if current panic stack fingerprint matches
- If so just grab dmesg and skip capturing dump (fast path)
- If not capture crash dump (multi-threaded makedumpfile) and reboot
- Use 16+ CPUs/threads to upload dump to object storage

Round Of Questions

- Do you customize your crash kernels at all?
- What's your memory reservation?
- Do you capture whole crash dumps or the output of a few diag commands?
- How many CPUs does your crash kernel use?
- Do you enable networking?
 - If so is it just for progress reporting or actual dump upload?
- Do you record and categorize/deduplicate failures in any way?

Crash Dump Fingerprints

- Based on panic stack and surrounding data (register values, error msg, etc..)
- A few different schemes to generate it

```
mapIterRange_IMPL+0x1d8/0x1e0 [nvidia] (P)
clientRefIterNext+0x114/0x180 [nvidia]
_handlePollMaskHelper+0xb4/0x2d0 [nvidia]
gpushareddataCtrlCmdRequestDataPoll_IMPL+0x88/0xf0 [nvidia]
resControl_IMPL+0x1c0/0x1d0 [nvidia]
serverControl+0x3b8/0x4a0 [nvidia]
_rmapiRmControl+0x440/0x640 [nvidia]
rmapiControlWithSecInfo+0xa8/0x150 [nvidia]
rmapiControlWithSecInfoTls+0x74/0xe0 [nvidia]
_nv04ControlWithSecInfo.constprop.0+0x80/0xa0 [nvidia]
Nv04ControlWithSecInfo+0x34/0x40 [nvidia]
RmIoctl+0x87c/0xbd0 [nvidia]
rm_ioctl+0x64/0x430 [nvidia]
nvidia_unlocked_ioctl+0x83c/0xa18 [nvidia]
__arm64_sys_ioctl+0xb68/0xc30
invoke_syscall.constprop.0+0x7c/0xf8
do_el0_svc+0x58/0xe8
el0_svc+0x168/0x208
el0t_64_sync_handler+0x44/0x108
```

Triaging Root-Cause Analysis with AI

- Occasionally hallucinates but keeps getting better
 - Currently a blocker for using it for failure categorization/deduplication
- Provided a good set of prompt, the more data we explicitly give it the better
 - System resource logs, source code, bug trackers, mailing lists, build manifests/artifacts
- Trade-offs:
 - Too much vs too little guidance
 - Models & Effort vs Cost
 - When to stop vs go on?

Round Of Questions

- Do you use AI at all for root-causing or triaging issues?
- What data & sources do you provide it? What are your prompts like?
- What are your constraints when using it? Cost? Node resources?
- Do you have automation around its usage?



Thank You!

Extra-Topic: OOB Introspection

- BMCs & Rack Managers give us a lot of hardware data on-demand
- Nothing like that for the running kernel
- Idea: DPU Observability (drgn through RDMA)
 - Main issue: security
- Question:
 - Does anyone do this in production? If so, how?