Building an Operating System/Runtime layer for Exasacale

Valentin Reis, **Argo** ECP project (PI Pete Beckman)

6th February 2019, Chameleon User Meeting

1. Argo
2. ChameleonCloud Use
   - Chameleon use case 1: Power control experiments
   - Chameleon use case 2: CI tasks
3. Feedback
   - What worked
   - What didn't
   - Wishlist

Goal: Design and prototype system-level software for exascale.

- Dealing with the new memory hierarchy,
- Node-level, container-based resource partitioning,
- Power management across the machine,
- Support for new HPC workloads (workflows, in-situ, streering)

Changing the role of an operating system

- Why multiplex/share resources if there are plenty?
- Why apply the same policies/abstraction to all applications?

Our solution

- Containers as a unified interface to OS services
- Give users exclusive access to partitioned resources
- Support HPC workloads through HPC-specific, user-activated features

Classic containers features

- Process namespacing + chroot
- Limited resource abstractions (QoS focused)
- Blind to actual hardware (topology, advanced features)

These are not adapted to HPC:

- Can create additional noise (extra services: sshd)
- No support from vendors (MPI, GPUs, high speed networks).

Argo NodeOS has *compute containers*, which give users the ability to reconfigure the node for their needs.

Compute containers features

- No namespacing or chroot: node is single user
- Exclusive use of partitioned resources
- MPI compatibility
- Apply scheduling policies on whole container
- Load & configure kernel modules

How it works

Container Scheduling

Take resource request + configuration as input, map it unto node.

Application Launch

Apply process-wide policies, fork directly inside container

```
{
  "acKind": "ImageManifest",
  "acVersion": "0.6.0",
  "name": "test",
  "app": {
    "isolators": [
      {
        "name": "argo/scheduler",
        "value": { "policy": "SCHED_OTHER", "priority": "0" }
      },
      {
        "name": "argo/container",
        "value": {
          "cpus": "24",
          "mems": "1"
        }
      },
      {
        "name": "argo/perfwrapper",
        "value": {
          "enabled": "1"
        }
      }
    ]
  }
}
```

Figure: Example of a container spanning 24 hardware threads that monitors power and CPU performance counters.

Chameleon use case 1: Power control experiments

We are currently performing dynamic power control experiments using containers:

Needs

- Baremetal instances, small reservation lengths (1 day)
- Multiple architectures (Haswell, Skylake, ect).
- ~~Custom Kernel~~ Root access (cgroups).
- Mix of interactive sessions and batch experiments.

We have a Gitlab-based CI pipeline that integrate multiple Argo components.

Needs

- Mix of Baremetal and KVM instances. The KVM part works, but Baremetal part is hard to put in place due to failure rates.
- Needs at least one baremetal flavor.
- ~~Custom Kernel~~ Root access (cgroups).
- Gitlab-runner and a Nix binary.

Power experiments

We run our power experiments using Baremetal deployed CC-Centos7 images.

Custom KVM images.

We have recipes for building cloud-init enabled openstack compatible images. These turned out to work seamlessly on the chameleon KVM cloud.

Various service quality issues.

- Failing nodes/IP binding/not enough nodes
- Non-atomic API calls for deployment
- Ticketing system error messages ("unauthorized" landing page on submission)

Our local solutions:

- Maitain a list of failing node IDs
- Switch to the CLI API (more responsive than the web GUI)

Conclusion: our Chameleon feature wishlist ☺

1. **Allow tweaking BIOS parameters** for memory management experiments on skylake nodes.
2. **Atomic API calls for deployment**
3. Automatically flag failing nodes.
4. Add fairness in scheduling policies?
5. (minor) Support for custom OS deployment on Baremetal.