Argo NodeOS: Toward Unified Resource Management for Exascale

Swann Perarnau*, Judicael A. Zounmevo*, Matthieu Dreher*, Brian C. Van Essen‡, Roberto Gioiosa†, Kamil Iskra*, Maya B. Gokhale‡, Kazutomo Yoshii*, Pete Beckman*
*Argonne National Laboratory. {swann, mdreher}@anl.gov, {iskra, kazutomo, beckman}mcs.anl.gov, zounm@linux.com
†Pacific Northwest National Laboratory. Roberto.Gioiosa@pnnl.gov
‡Lawrence Livermore National Laboratory. {vanessen1, maya}@llnl.gov

***Abstract*—Exascale systems are expected to feature hundreds of thousands of compute nodes with hundreds of hardware threads and complex memory hierarchies with a mix of on-package and persistent memory modules.**

In this context, the Argo project is developing a new operating system for exascale machines. Targeting production workloads using workflows or coupled codes, we improve the Linux kernel on several fronts. We extend the memory management of Linux to be able to subdivide NUMA memory nodes, allowing better resource partitioning among processes running on the same node. We also add support for memory-mapped access to node-local, PCIe-attached NVRAM devices and introduce a new scheduling class targeted at parallel runtimes supporting user-level load balancing. These features are unified into *compute containers*, a containerization approach focused on providing modern HPC applications with dynamic control over a wide range of kernel interfaces. To keep our approach compatible with industrial containerization products, we also identify contentions points for the adoption of containers in HPC settings.

Each NodeOS feature is evaluated by using a set of parallel benchmarks, miniapps, and coupled applications consisting of simulation and data analysis components, running on a modern NUMA platform. We observe out-of-the-box performance improvements easily matching, and often exceeding, those observed with expert-optimized configurations on standard OS kernels. Our lightweight approach to resource management retains the many benefits of a full OS kernel that application programmers have learned to depend on, at the same time providing a set of extensions that can be freely mixed and matched to best benefit particular application components.

I. Introduction

Exascale systems are expected to feature hundreds of thousands of compute nodes with hundreds of hardware threads and complex memory hierarchies with a mix of on-package and persistent memory modules. The International Exascale Software Project [1] identified a number of challenges that need to be addressed on such systems. On the operating system (OS) side, the roadmap advocates that interfaces and support for new types of memory must be developed. Additionally, OS software should provide explicit control, from user space, of the resources available on the compute nodes. At the runtime level, parallel languages should transition from straightforward fork-join parallelism to asynchronous overdecomposed approaches, with architecture- and topology-aware load balancing performed by the runtime itself.

Following this roadmap, we argue that the role of a multitasking OS is transitioning from managing access to shared resources on the node (CPU, memory, NIC, etc.) using multiplexing techniques such as time sharing and swapping, which may be disruptive to many HPC workloads, to coarsely *partitioning* those numerous resources and bundling them together in an integrated fashion through a unified interface—*containers*. Lightweight runtimes [2], [3], forming part of comprehensive parallel programming frameworks, will then be given exclusive control of resources to perform custom redistribution according to their knowledge of the application and its inner parallelism. Such an approach ensures a more deterministic execution and noticeably lower overheads.

Furthermore, the increasing resource density on HPC nodes, combined with the growing relative cost of internode communication, provides a strong motivation for new kinds of HPC applications. Increasingly, a single job consists of a computation component with a data analytics or visualization component running alongside it. Care must be taken to ensure that such coupled components, which can have very different requirements for utilizing node resources or system services, actually benefit from running in close proximity to each other instead of grinding to a halt because of unintended interference.

A. Example Workload

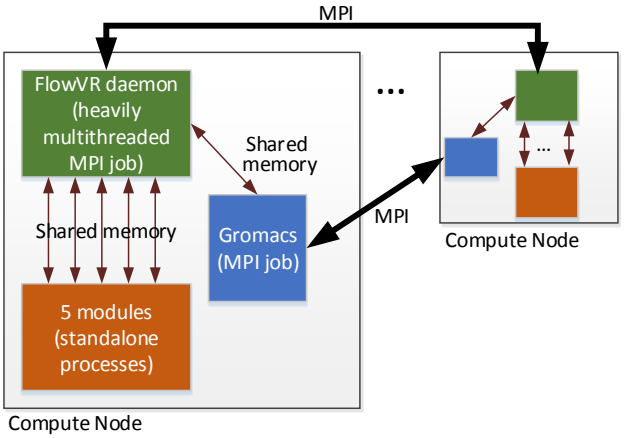


Fig. 1: Process interaction of a coupled application.

Figure 1 presents an overview of a complex application workflow and the interaction between its processes. The sim-

ulation application in the workflow is Gromacs [4], a standard molecular dynamics simulation package in the biology community. In biology, visualization is a key tool for understanding the functions of molecular systems. Isosurface extraction using the Quicksurf [5] algorithm is being used here for rendering. This algorithm is implemented in situ [6] to avoid the high cost of I/O between components.

The coupling is managed by FlowVR, an in situ middleware designed for building asynchronous workflows. The Gromacs simulation code runs multiple processes on each node, in MPI-only mode, and has been modified to extract atom positions at run time rather than writing them to a file. The in situ visualization component is a pipeline consisting of five sequential processes on each node. Three of them are compute modules: (1) distributing the atoms in a regular grid, (2) computing a density value for each cell based on the atoms in the cell and its neighborhood, and (3) computing a marching cube on the density grid. The two remaining steps perform redistribution of atom positions. Data exchanges between modules are performed by using a shared-memory space managed by the FlowVR daemon hosted on each node. If the sender and receiver modules are on the same node, the daemon simply passes a pointer to the receiver; otherwise, it uses MPI to send the data to the remote daemon hosting the destination module. The daemon is heavily multithreaded, consisting of four internal threads plus a thread for each module running on the node; none of them are computationally intensive.

Correct placement of application processes on the node is critical to obtaining optimal performance. The five in situ analytics processes together require at most 20% of the CPU cycles of a single core, but they must be kept apart from the Gromacs processes, which are highly sensitive to perturbations.

This workflow exemplifies the complexity of future production jobs on exascale systems from a resource management standpoint. Inside a single batch job, three types of processes share the same nodes, with distinct requirements and optimization opportunities. First, the simulation component (Gromacs) is a typical MPI application, capable of using OpenMP internally. As parallel programming runtimes are transitioning to user-level threads and internal dynamic load balancing [2], [3], this group of threads could benefit from an improved system scheduling policy. Second, visualization stages are single threaded and involve low overhead but must be kept apart from the simulation for performance reasons. At the same time, the communication between the simulation and the visualization requires shared memory access and might also use a shared NVRAM device in the future. Third, the FlowVR daemon must share memory and cores with both Gromacs and the visualization and is implemented through Pthreads. Fourth, system services might run on the nodes and should also be isolated from the user processes. With standard Linux interfaces, satisfying all these requirements is tedious at best and prone to errors.

B. Contributions

Driven by the growing complexity of HPC workloads, we outline a containerization approach focused on providing modern HPC applications with a unified interface for resource partitioning and other kernel features, which is also compatible with popular container frameworks. Built on top of the Linux kernel, our *compute containers* simplify the management of complex workloads and provide users with a single API to further customize the environment of each component.

We improve the Linux kernel and our container approach following three complementary axes. First, we extend the memory management of Linux to be able to subdivide NUMA memory nodes. Second, we add support for memory-mapped access to node-local, PCIe-attached NVRAM devices. Third, we introduce a new scheduling class targeted at parallel runtimes supporting user-level load balancing. We evaluate these features using a set of parallel benchmarks running on a modern NUMA platform. The experiments show that our NodeOS enables better utilization of the hardware, with less effort than would be required in a fully "manual" approach.

The remainder of this paper is organized as follows. Section II provides an overview of the NodeOS architecture and our approach to containerization. Section III focuses on the resource-partitioning aspect. Section IV presents our work on managing PCIe-attached NVRAM. Section V discusses improvements to the kernel task scheduler and other jitter mitigation techniques. Section VI outlines related work. Section VII presents the conclusion and planned future directions for this work.

II. Compute Containers as a Unified Interface to Kernel Facilities

We introduce here the concept of *compute containers* to provide integrated control over individual hardware and software resources on each node of an exascale machine. Our compute containers have a different purpose from that of popular containerization frameworks such as Docker [7]. Our goal is to accommodate divergent needs of different components of complex applications. As an example, Figure 2 shows a node configured with three compute containers. Each application component has a partition of node resources (CPU cores, memory) allocated to it (see Section III). In one container, a portion of the memory is used as a DRAM cache for on-node NVRAM to accommodate out-of-core computations (see Section IV). Our containers can use different task-scheduling policies; traditional HPC workloads as well as those using lightweight, user-space threading ("Concurrency" in the figure) can benefit from an HPC-specific scheduler (see Section V), whereas highly oversubscribed workloads (e.g., for hiding the NVRAM access latency) tend to run better under the default Linux CFS scheduler.

A. Containers for Future HPC Systems

Our compute containers differ from mainstream containers (see Section VI) in several ways. We list here the aspects of our

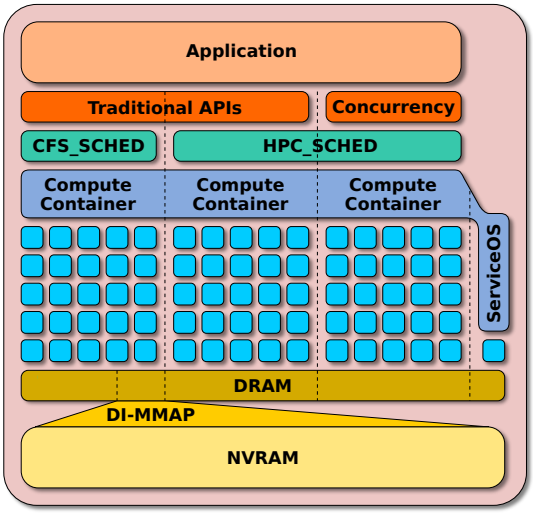


Fig. 2: Example configuration of the Argo NodeOS.

solution that reflect our focus on providing a containerization solution for HPC platforms.

No namespaces: our containers do not use namespaces or `chroot`. As is common in HPC, we expect all the user processes on a compute node not only to belong to the same user but also to be part of the same job and thus communicate with one another. Erecting barriers in the form of separate process or file system namespaces would be counterproductive to this goal. This approach also removes considerable complexity and overhead from container management.

MPI support: we provide a lightweight solution to launch MPI ranks into their own containers. A common solution to run MPI processes inside mainstream containers is to run an `ssh` daemon inside the container and have `mpirun` connect to it. Instead, we provide our own MPI launcher that enters the container during the launch sequence and exits it afterward. This avoids additional system noise for each contained MPI rank. We also support remapping the MPI ranks binding to the cores available in the target containers.

Scheduling policies: our containers allow a user to specify that all the contained processes should be under a configurable scheduling policy. We apply this scheduling policy when creating new processes inside containers, and it is inherited automatically by any Linux process spawned within, as per the regular Linux forking semantics. If necessary, new processes can change their scheduling policy from inside the container.

Module loading and configuration: our containers can request the loading and configuration of custom modules, to allow platform-specific features to be accessed dynamically inside containers. We use this feature for our support of DI-MMAP (see Section IV).

We also provide partitioning of node resources, as explained in the next section.

B. Implementation

From a user's point of view, a compute container is described by using a file following the standard container description format APPC [8]. This format allows for implementation-defined extensions; we provide our additional features through those extensions. Thus, a user can specify in the container description the amount of resources required by the application, indicate whether a particular scheduling policy should be in place, and identify what subsystems such as a high-performance DRAM cache for node-local NVRAM are to be configured. Figure 3 shows an example of such a container manifest.

```
{
    "acKind": "ImageManifest",
    "acVersion": "0.6.0",
    "name": "mycontainer",
    "app": {
        "isolators": [
            {
                "name": "scheduler",
                "value": {
                    "policy": "SCHED_HPC",
                    "priority": "0"
                }
            },
            {
                "name": "container",
                "value": {
                    "cpus": "48",
                    "mems": "1"
                }
            }
        ]
    }
}
```

Fig. 3: Example of a container spanning 48 hardware threads and using the HPC scheduler.

Our implementation is split into several components. First, the node provisioning: as we explain in Section III, we partition off resources for system services. In order for this partition to contain all the system services, including the dynamically created processes, either the system init process must be modified or all existing processes must be moved to this partition later in the lifetime of the node but before an application is launched on it.

Second, the container scheduling: the compute container matching the high-level description provided by the user must be created before the application can run. We call this process *scheduling* a container. The container description lists the size of necessary resources but not which resources to use. This issue is handled by our container manager, using information about the system topology to select appropriate resources. Additionally, the container manager sets the selected scheduling policy and performs the necessary steps to set up a DRAM cache for node-local NVRAM on behalf of the application. This component also allows privileged operations (for the ServiceOS) and nonprivileged ones (user actions) for manipulating compute containers. It reuses the Linux uid mechanisms to apply ownership to partitions, thus preventing undue task and resource migrations or deletion by other uids. There is also a comprehensive reporting tool for node configuration.

Third, the application launching: we leverage the fact that any process created on Linux inherits the container of its parent to fork the application directly inside its container. All

processes and threads created by the application later will also be constrained. In the case of MPI applications, we ask the user to provide a container manifest to `mpirun` instead of an application executable to run. The manifest is then distributed to all nodes, where local containers are scheduled before executing the application. Since our compute containers do not modify the environment of the application or its view of the file system, most features work without change.

III. Partitioning

We extend the Linux kernel with additional resource control capabilities to maximize resource utilization with HPC workloads. We take advantage of the control groups (*cgroups* [9]) resource isolation infrastructure that is a foundation of most containerization frameworks, ours included.

The resources are first partitioned during node bootup. A small subset of CPU and memory resources is bundled into a *ServiceOS* partition (see Figure 2), which is subsequently populated with system services as well as non-HPC tasks running on the node. The bulk of the resources is thus left unused so as to be allocated to compute containers at job launch time in order to host the HPC application processes. Partitions ensure a perfect separation of user-space processes, although some kernel activities may still use resources allocated to the compute containers.

A. CPU

The CPU resources are partitioned by using the existing `cpusets` resource controller, which associates a set of CPU cores (or hardware threads, to be more precise) with a cgroup. To ensure the most predictable run-time behavior, we partition CPU cores such that each is allocated to only a single partition. The ServiceOS is normally allocated a single CPU core (typically consisting of multiple hardware threads). The remaining cores can all be allocated to a single compute container or, at the request of the application, be divided among multiple containers to accommodate complex workflows (see Section I-A).

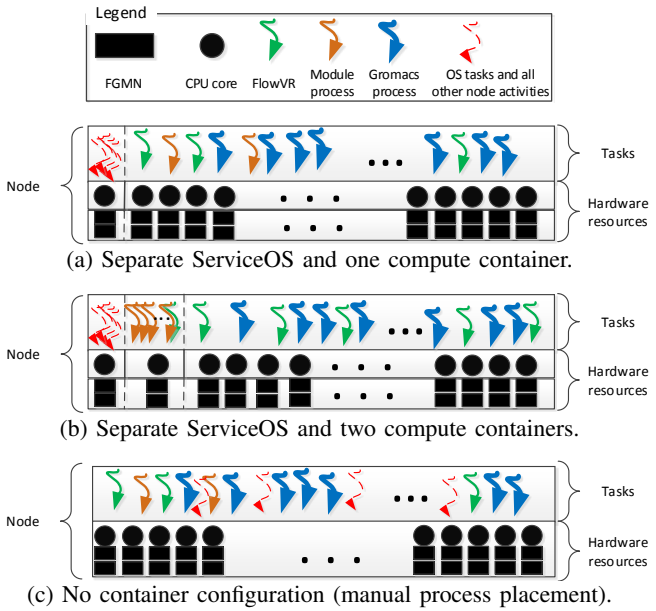
B. RAM

Partitioning the system memory is a more complex operation. In principle, the `cpusets` controller can also associate a memory node with a cgroup; however, the granularity is at the NUMA node level, which is too coarse for our purposes (ServiceOS does not typically need a whole NUMA node, and not all systems are NUMA in the first place). Linux does offer a `memory` controller that can set a limit at byte granularity; but it lacks control over physical memory location, making it an unattractive option in the face of deepening memory hierarchies consisting of multiple tiers of DRAM in addition to node-local, PCIe-attached NVRAM.

To solve this problem, we implemented finer-grained memory nodes (FGMNs). Physical NUMA nodes are partitioned into arbitrary logical blocks at a user-defined granularity, based on a specification supplied at boot time. These partitions are presented to the rest of the system as separate NUMA nodes, enabling their use by the `cpusets` controller. FGMNs are a reimplementation of the *fake NUMA* feature of the standard Linux kernel, with significant enhancements in terms of flexibility and robustness. We normally allocate under 5% of memory to the ServiceOS and leave the rest to be allocated to the compute containers. The kernel is not bound by memory partitioning and can satisfy its internal dynamic allocations (such as the page cache) with memory from any node. The cache of DI-MMAP, however, is NUMA-aware and so can be constrained as desired (see Section IV).

C. Evaluation with a Complex Workflow

We are using the Gromacs workflow outlined in Section I-A. It is run on nodes consisting of dual-socket, 12-core Intel Xeon E5-2670 v3 processors, 128 GiB of RAM and 10 Gbps Ethernet NIC. Hyperthreading is enabled. The nodes are booted with 48 FGMNs of approximately 2.6 GiB each. Figure 4 outlines the tested container configurations.



(a) Separate ServiceOS and one compute container.

(b) Separate ServiceOS and two compute containers.

(c) No container configuration (manual process placement).

Fig. 4: Different node configurations for the coupled application.

Because of constraints in its internal decomposition algorithm, Gromacs works well only with certain process counts. For consistency, we opted to run it with 21 processes in all configurations, even if in some cases that leaves an idle core. Data is extracted every 10 iterations of the simulation. In the first configuration (Figure 4a), a ServiceOS is confined to a single core (two hardware threads) and two FGMNs while the application workflow as a whole is executed in a single compute container using the rest of the resources. This is the default configuration for conventional SPMD HPC jobs, but we do not expect it to provide a sufficient isolation for Gromacs inside the overall workflow. The second configuration (Figure 4b) is meant to address that; it creates a separate compute container for the in situ visualization and

data analysis (VDA), with one CPU core and two FGMNs, leaving the rest for the Gromacs simulation container. Because the FlowVR management daemon communicates with all the components of the workflow, we allow it to run unconstrained. While the tasks running in each container allocate memory from their assigned FGMNs, there is purposely no restriction for cross-container memory access, the reason being the need to allow shared-memory communication. The third configuration (Figure 4c) reproduces the second one manually by setting affinities for the processes on a vanilla node with no partitioning; it is meant to serve as a baseline when assessing the overhead introduced by NodeOS configurations.

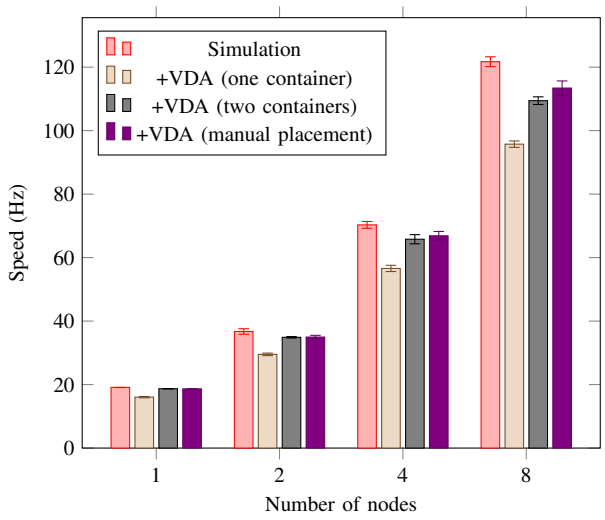


Fig. 5: Performance of the coupled application under different node configurations.

We ran Gromacs and the in situ Quicksurf pipeline with the three configurations described above on up to eight nodes; the results are shown in Figure 5. Performance is reported in `Hz`: the amount of iterations the simulation manage to perform in a given time period. Also included in the figure is a run labeled *Simulation* that gives a performance baseline for Gromacs modified to use FlowVR but without the in situ pipeline running. The *+VDA (one container)* run denotes the configuration from Figure 4a. Each of the Gromacs processes is pinned to a single core. The container has an extra core where the in situ processes could run, but the kernel scheduler is not constrained to run them only there, which leads to a 21% increase in Gromacs execution time compared with the baseline in the worst case at eight nodes, as well as an increased performance variability. The *+VDA (two containers)* run denotes the configuration from Figure 4b. In this case, the performance impact of the concurrent execution of the in situ modules is reduced to only 2.2% over the baseline at one node and 10% at eight nodes, an improvement of 16.45% at one node and 11% at eight nodes over the one-container run. The performance penalty indicated by the two-container run over the baseline is due to an increase in the synchronization rate between Gromacs and FlowVR. While a single synchronization has a fixed cost, as the simulation speed improves, more synchronizations occur in the same time period. Note that we deliberately chose a high output frequency to amplify the perturbations on the system, which explains the 10% cost over baseline at eight nodes; a normal production run would output data every 1,000 iterations or less. The *+VDA (manual placement)* run, which corresponds to the configuration from Figure 4c, reproduces the same process placement as the two-container run, but using the `taskset` command manually rather than using containers. The performance is essentially the same up to four nodes, but a slight overhead of 3% arises at eight nodes. We limited the experiments to eight nodes because the chosen high output frequency overflows the in situ pipeline with more nodes. These experiments demonstrate that using the containers can free the user from the burden of process placement, for a limited performance penalty.

IV. Incorporating SSD

The preceding sections addressed resource management of CPU cores and associated main memory regions. We now discuss extending resource management to data accessed from node-local SSDs, which are included in many proposed exascale node architectures.

To efficiently use node-local, PCIe-attached NVRAM SSD, we integrated the data-intensive memory-map (DI-MMAP) runtime into NodeOS. DI-MMAP has been optimized for data-intensive applications that use a high-performance PCIe-attached SSD for extended memory or for accessing persistent data structures via the memory-map interface. Prior work [10] demonstrated that DI-MMAP can outperform the traditional Linux memory-map by significant margins for memory-constrained, data-intensive applications. We integrated DI-MMAP into NodeOS to provide similar performance benefits, as well as to provide additional tuning knobs for controlling the effects of I/O within the HPC node.

DI-MMAP has been integrated with containers in order to give the user maximal control over allocation and management of the HPC compute node persistent memory resources, particularly in their interaction with CPU cores and main memory. DI-MMAP provides an explicit buffer and page management strategy that has been optimized for out-of-core memory-mapped I/O usage that requires page eviction on almost every page fault. In order to access the DI-MMAP buffer, users first have to ask for its allocation using virtual configuration files (sysfs), and can control the memory location of the buffer using `numactl`. Our compute containers execute those steps directly during the launch of a container with the appropriate configuration.

In this section we present a series of experiments showing the performance of DI-MMAP with and without compute containers. The experiments were run on an Intel CPU E7-4850 @2.00 GHz (quad-socket Westmere) with 512 GiB memory, 40 cores (80 threads), PCIe Gen 2 x16 slot, with a 1.2 TB Intel SSD (P3700 NVMe card, PCIe Gen3 x4). This node uses RHEL7 with a 4.1.3 Linux kernel. Our first experiment

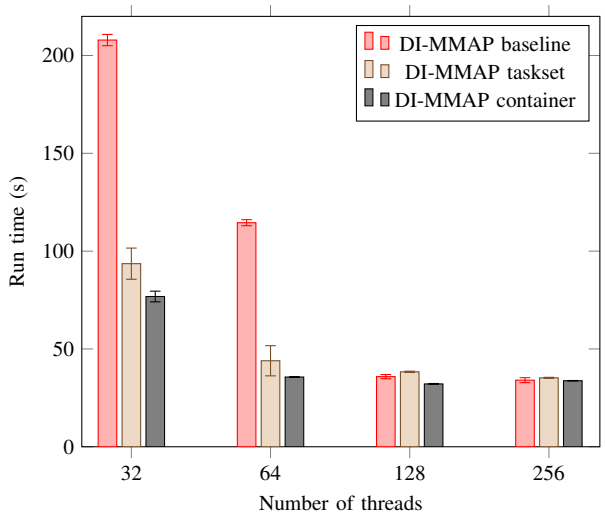


Fig. 6: Comparison between run times of LRIOT accessing a file through DI-MMAP. Application and buffers: unrestricted, limited to a single socket with `taskset`, and limited to a single socket with containers, respectively.

used a synthetic I/O benchmark—the Livermore Random I/O Toolkit (LRIOT) [10] designed to test I/O to high-performance storage devices, especially PCIe-attached SSDs. The benchmark generates tests that combine multiple processes and multiple threads per process to simulate highly concurrent access patterns. Sequential and random patterns can be tested at user-specified read-to-write ratios using memory-mapped, direct, or standard I/O.

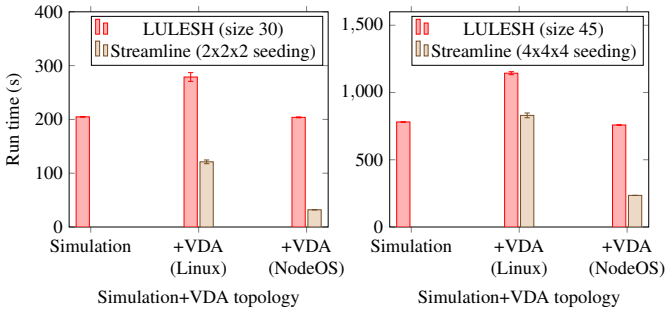
Figure 6 reports the performance of a single LRIOT process mapping a 384 GiB file from the SSD using DI-MMAP. The number of threads per core was varied from 32 to 256 (x axis). DI-MMAP used the buffer size of 8 GiB, preallocated when the kernel module was loaded. This configuration is consistent with the use case of data-intensive applications using all available memory for dynamic data structures in their heaps. The run time to complete 12,582,912 4 KiB read ops is shown on the y axis. Three setups are reported. For the baseline, the benchmark and DI-MMAP run unconstrained across the whole machine. In the second setup, LRIOT is constrained to a single socket with `taskset`, while the DI-MMAP buffer is still allocated across all NUMA nodes. The third test uses compute containers to constrain cores and associated memory regions to a single NUMA node for LRIOT and DI-MMAP. It shows DI-MMAP with higher performance under compute containers than using `taskset` or the unconstrained case. DI-MMAP under containers is 20% faster for 32 threads, 23% for 64 threads, 19% for 128, and 4% for 256, than with `taskset`. The diminishing differences between the 3 experiments are due to the NVMe device reaching saturation (maximum number of IOPs it can service).

These results show that for highly concurrent, threaded applications with read-heavy I/O (e.g., analyzing a visualization dataset generated by simulation or querying a database), DI-MMAP with compute containers is significantly faster than with standard `taskset`; thus, compute containers provide a convenient isolation mechanism to improve performance.

The LRIOT benchmarks tested highly multithreaded, read-only, memory-mapped I/O. To extend testing to use cases more representative of future HPC workflows, we ran these scenarios on a workload with an HPC miniapp and an analysis benchmark. The LULESH miniapp was used as the simulation code. In parallel, we ran a streamline tracing [11] visualization and data analytics application that traverses an out-of core dataset, data from a 3072^3 Rayleigh-Taylor (RT) instability simulation from LLNL. The simulation evolves two fluids mixing, creating a turbulent mixing layer that yields a complex flow field. The streamline tracing code searches for flow stream lines in the flow field. The code has been adapted to access the data set through memory-mapped I/O.

We first measured the performance with LULESH running by itself on the entire compute node, as would be done in a traditional simulation workflow. The next case had the two applications run concurrently without any restrictions under standard Linux, with the streamline benchmark using standard memory-mapped I/O to access files on the SSD. Then we used two compute containers, one for each application, partitioning the available node resources between the two. This setup also made use of DI-MMAP for streamline tracing. In order to mimic the interaction of coupled codes in this setup, the applications were launched concurrently and coarsely tuned to have run times similar to a first-order approximation.

The LULESH simulation was configured to run using 64 processes, one thread per process, on a $size = 30$ (*Modest test size*) and $size = 45$ problem (*Large test size*). There were $size^3$ elements per domain and one domain per process. The streamline code ran as a single process with 64 threads, each tracing a streamline using uniform seeding. The streamline length was set to 4096, and seed points were uniformly sampled from each 3-D region. The sampling density was either 2x2x2 per region for the modest test size or 4x4x4 per region for the large test size.



(a) Modest test size. (b) Large test size.

Fig. 7: LULESH + streamline VDA.

The results of the experiment are summarized in Figure 7. The leftmost bar shows the run time of LULESH by itself. The next set of bars is the run time of each application as they run concurrently in unconstrained Linux without core

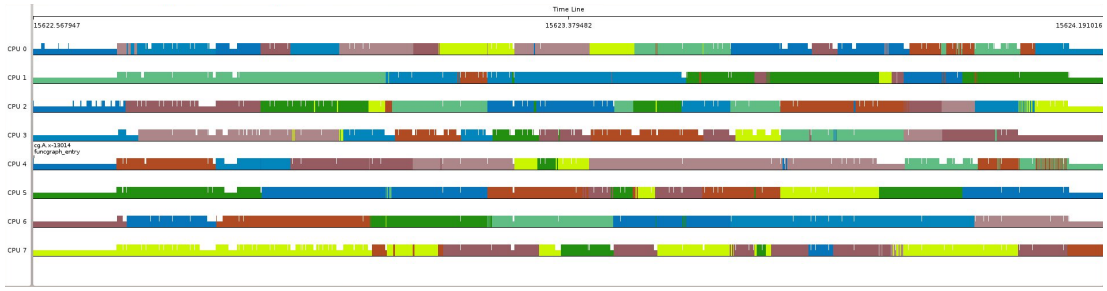


Fig. 8: NPB CG execution trace. Each color in the trace represents a different task while vertical bars represents kernel activities. The figure shows that (1) tasks are generally migrated from CPU to CPU even if there is an equal number of CPUs and tasks and (2) kernel activities often interrupt computation.

affinity or regard to placement. The rightmost set of bars shows performance when the two applications are run in two separate containers. As the figure shows, performance of LULESH is greatly reduced when the visualization and data analysis streamline code runs with it in standard Linux, while it is the same or slightly better when each application is constrained to a separate container. LULESH performance degrades by 36.1% and 46.4% over the baseline with standard Linux for the modest and large experiments, respectively. Running in a container prevents the streamline code from taking resources from LULESH. Thus, the simulation performance is not impacted by having a concurrent "in situ" application running on the compute node. Note that the performance of the streamline-tracing VDA application with DI-MMAP (in NodeOS) is 3.78× and 3.53× faster than the standard Linux environment for the modest and large tests, respectively. This performance improvement for the streamline-tracing VDA application is due largely to improvements DI-MMAP offers beyond standard mmap that were previously published [10], with some contribution due to the performance isolation of containers illustrated in Figure 6.

V. Scheduling

In user mode, the CPUs and memory resources are partitioned by using control groups and FGMNs, respectively. In kernel mode, however, a single Linux kernel image is still shared by all the cores in the systems. Thus, when an application thread enters the kernel, either voluntarily through a system call or involuntarily through an interrupt, the HPC application effectively competes with the other user and kernel daemons in the system to access hardware and software resources. For example, an application thread can spin on its runqueue lock in a timer interrupt as long as the lock is held by another function (e.g., a process migration or load balancing).

Full-fledged OS kernels such as Linux can introduce OS noise during the execution of HPC applications [12]–[14]. In our approach, we limit the number of interruptions and cycles spent in kernel mode to the detriment of HPC applications. To this extent, we need to limit process preemptions as well as the execution of kernel activities that are not directly related to the HPC application.

Figure 8 shows the execution of a conjugate gradient OpenMP application on a system with 8 compute cores using CFS. The trace was obtained from `ftrace` [15], a lightweight kernel-tracing system available in recent versions of the Linux kernel. In the trace, the *x* axis shows the execution time, each row represents a processor core, and each OpenMP thread is assigned a specific color. As we can see, application threads are continuously migrated from CPU to CPU, depending on the particular task load of each processor at the time load balancing is performed. Note that although in this experiment we ran 8 application threads on an 8-core system, not all threads are necessarily runnable at the same time. In fact, the system load may show an imbalance because a thread is blocking on a long-latency operation or because a user or kernel daemon has woken up. When the load becomes imbalanced, the CFS load balancer will attempt to redistribute the load by migrating tasks. High bars in each CPU row denote kernel activities performed while the application threads are running.[1]

We modified the Linux kernel to provide performance isolation *and* increased throughput. We developed a new task scheduler (`HPC_SCHED`) that simplifies the scheduling of HPC application processes and threads. This new scheduler is an OS kernel component and orthogonal to user-level containers. As such, the scheduler can be used in conjunction with containers (i.e., the HPC application running in a computing container could leverage our new scheduler) or in isolation (in case containers are not used). The advantages of the new scheduling algorithm are threefold. First, HPC application tasks are scheduled by a specific scheduler and do not increase the CFS load. Thus, HPC tasks are not considered during CFS bookkeeping operations, such as the load balancing in the previous example. Second, we prioritize the `HPC_SCHED` scheduling class over the regular CFS scheduling class. Thus, tasks in the CFS class cannot preempt HPC tasks and can run on a CPU only if no runnable HPC tasks are assigned to that CPU at the time. Third, `HPC_SCHED` implements a set of scheduling algorithms that are designed for HPC applications. Specifically, `HPC_SCHED` can be configured with the following options.

[1]Because of pixel resolution, threads seem to be in kernel mode for long intervals. However, zooming in the trace reveals many independent, short kernel activities, such as timer interrupts, that appear as one block in the trace.

User. The OS does not perform any assignment of threads to CPUs; the user-level runtime is in charge of setting the affinity of each application thread. This mode of operation is intended for intelligent runtimes that implement their own scheduler.

RoundRobin. Tasks in the `HPC_SCHED` scheduling class are assigned to CPUs in a round-robin order. If the number of tasks in the `HPC_SCHED` class is larger than the number of available CPUs, some CPUs will be oversubscribed.

TopologyAware. Tasks are assigned to CPUs in a way that maximizes performance by reducing hardware resource contention, typically by employing a breadth-first approach.

In all cases, the scheduling of HPC tasks on the same CPU is supposed to be cooperative; in other words, tasks voluntarily release the CPUs to other tasks to offer them a chance of making progress. Currently, `HPC_SCHED` does not implement time sharing and preemption among HPC tasks in the same run queue. This design choice follows the general execution strategy of one thread per CPU traditionally employed in HPC environments. Notice that even when using the RoundRobin and TopologyAware policies, `HPC_SCHED` will still honor user-defined CPU affinity and will not schedule a task on a CPU that is not set in the task's CPU affinity mask. This design ensures seamless integration with the rest of the NodeOS components, especially containers.

We also analyzed many kernel control paths that are normally executed on each CPU to ensure the correct functioning of the system and a fair sharing of hardware resources among the active processes. Many of these activities are unnecessary in HPC environments, where compute nodes are typically assigned to a single user. We identified the most common kernel activities that can be removed without impacting the correctness of running applications. We traced several applications and ranked the most common kernel activities and those with the largest variance, which have the highest probability of being harmful. To limit the number of code modifications, we followed an iterative approach, removing unnecessary kernel control paths until we achieved satisfactory performance. Removing just a few activities, such as the CFS load-balancing operations, already provides considerable advantages. Currently, we do not disable the local timer interrupt on CPUs running HPC tasks but instead employ a full dynamic timer (tickless) system.

We performed our evaluation on a dual-socket AMD Opteron 6272 equipped with 16 cores per socket, 64 GiB of RAM divided in four NUMA domains, and an Infiniband interconnection network. We analyzed the impact of `HPC_SCHED` when running parallel OpenMP applications. We ran 32 OpenMP threads, one per CPU core. We selected several NPB kernels (BT, CG, LU), HPC applications (LULESH), and data analytics workloads (Graph 500) and compared `HPC_SCHED` with the standard CFS scheduler and with execution when application threads are statically bound to cores ("CPU Affinity"). This execution was achieved by setting the `GOMP_CPU_AFFINITY` environment variable used by the OpenMP runtime. For Graph 500 the speedup

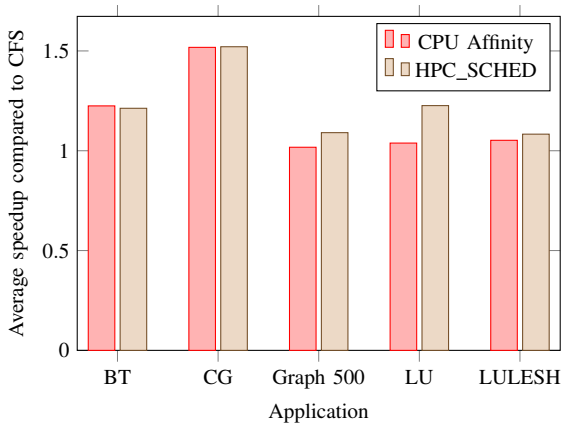


Fig. 9: Speedup of OpenMP with statically set affinity and `HPC_SCHED` compared with standard CFS.

was computed in terms of TEPS (harmonic mean across 64 searches). Figure 9 shows results in terms of speedup with respect to the execution with CFS. The graph shows that applications with good locality (BT and CG) benefit primarily from reusing data in the processor caches and are sensitive to unnecessary process migrations often performed by CFS. For these applications, statically setting the CPU affinity of each thread provides considerable performance improvements, up to 1.5× for CG. Both `HPC_SCHED` and the static CPU affinity achieve this performance. In other cases, however, simply statically binding application threads to cores does not provide much benefit, if any. This situation happens for applications with poor locality or whose dataset does not fit in the processor caches. As we can see in Figure 9, `HPC_SCHED` achieves considerably better performance for Graph 500, LU, and LULESH—respectively 1.09×, 1.20×, and 1.10×—while statically binding threads to cores in the OpenMP runtime achieves measurable performance improvements only for LULESH (1.06×) and marginally for LU (1.03×).

Interesting, and perhaps counter-intuitive, `HPC_SCHED` also provides performance improvements for data analytics. In fact, as explained above, `HPC_SCHED` not only prevents application threads from moving from one CPU to another (increasing cache locality) but also favors HPC tasks over CFS tasks and reduces the amount of OS noise on the CPU running the HPC application tasks. These advantages still hold when using containers, as several user and kernel daemons may be running in the CFS scheduling class, and the CFS scheduler could decide to move threads from one CPU to another within the same container. The result is a higher responsiveness of the data analytics tasks that are able to find an available CPU as soon as they become runnable.

VI. Related Work

Custom OS kernels for HPC workloads have a long tradition. In particular, dedicated lightweight kernels have been researched for decades, going back to SUNMOS [16] and continuing over multiple generations [17]. More recent examples

include Berkeley's Tessellation [18] and Akaros [19], IBM's CNK [20] and FusedOS [21], RIKEN's McKernel [22], and Intel's mOS [23].

The bulk of HPC systems, however, run a customized version of a mainstream operating system, typically Linux. Research projects in this area tend to focus on memory management and task scheduling. Examples of the former include explorations of Shmueli et al. [24], ZeptoOS [25], and HPMMAP [26]. The issue of OS jitter, or noise, has also been extensively studied [12]–[14], [27]–[31], including work on improving the process scheduler [32].

Management of CPU resources through core specialization has been explored as well, in Linux (i.e., Cray's CNL [33]) and in many of the aforementioned lightweight kernels. The latter frequently feature a more extreme form of specialization, with multiple OS kernels running on a single node using hardware partitioning or virtualization (e.g., Palacios and Kitten [34]); Linux is typically one of them and is used for overall node management or to provide the necessary high-level functionality.

While drawing inspiration from this past research, in our work we have attempted to implement HPC-specific improvements in a minimalistic, sustainable fashion, taking advantage of the flexibility of the Linux kernel and its existing mechanisms such as the transparent huge pages, pluggable schedulers, and cgroups.

Containers were initially broadly adopted for hassle-free packaging deployment and as a replacement for hypervisor-type virtualization. They quickly gained popularity for their lightweight nature and their bare-metal performance [35], [36]. In HPC, container technologies are being studied as a dependency management and packaging solution, in NERSC's Shifter work [37] for example. Our containers are different from popular technologies such as Docker [7] and Rkt [38] in that we focus on hardware resource management, an effort that is distinct from but complementary to packaging. We intend to be compatible with existing technologies, starting with the use of the APPC container description standard.

Process placement and resource allocation are key problems for in situ application. Several strategies have been proposed to manage efficiently the computational resources. Initial implementations used synchronous function calls inside the simulation mainly for visualization [39], [40]. Other methods allocate dedicated cores [41] or dedicated nodes [42] to perform in situ computations. These strategies have a direct impact on the simulation and analytics performance. Hybrid systems such as FlexIO [43] and FlowVR [6] address the need for placement flexibilityby combining the three placement strategies in the same in situ workflow.

VII. Conclusion and Future Directions

In this paper we introduced the concept of *compute containers*, a containerization approach providing integrated control over individual hardware and software resources on HPC nodes of current and future systems. Containers provide an easy-to-use, integrated platform to manage the partitioning of CPU and memory resources, to configure memory-mapped access to node-local, PCIe-attached NVRAM devices, and to select process scheduling best suited to the particular workload characteristics.

We evaluated individual components on a set of benchmarks and miniapps, including LRIOT (designed to test I/O to high-performance storage devices), NAS Parallel Benchmarks, Graph 500, and LULESH. We observed out-of-the-box performance improvements easily matching, and often exceeding, those observed with expert-optimized configurations on standard OS kernels. For example, LRIOT with containers showed an improvement of up to 23% over a configuration limited using `taskset`, and `HPC_SCHED` demonstrated an improvement of up to 20% compared with a similar baseline.

We also performed more complex experiments using coupled codes, including a coupled application consisting of Gromacs simulation and Quicksurf in situ data visualization, as well as simulated workloads consisting of LULESH as a simulation component and streamline tracing as a data analysis component. Experiments with Gromacs confirmed that resource partitioning using containers removes from the user the burden of manual process placement, at a limited performance penalty of at most 3%. Experiments with LULESH coupled with streamline tracing using containers demonstrated excellent performance isolation of the simulation component.

Overall, the conducted experiments confirm the viability of our lightweight approach to resource management, which retains the many benefits of a full OS kernel that application programmers have learned to depend on, yet at the same time offers additional HPC-specific extensions. Rather than providing a constricted "one size fits all" environment so often seen with lightweight kernels, our extensions are essentially a toolset that can be freely mixed and matched resulting in an abundance of possible node-local or even process-local configurations to best benefit particular application components or workflow management middleware. We consider the flexibility this provides to be a critical aspect of our approach.

We hope to continue this work toward the first exascale systems and beyond. We want to extend the set of node resources managed by containers (interconnect would be the first obvious example) and continue to tune components to emerging integrated workflows. Testing our approach at larger scales is an important goal that we hope to realize as the set of suitable coupled application codes keeps growing. We are also designing additional APIs for applications to change dynamically the resources allocated to their containers, to support long running jobs with phases of varying workload intensity.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357. Part of this work was performed under

the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344.

REFERENCES

[1] J. Dongarra, P. Beckman *et al.*, "The International Exascale Software Project Roadmap," *International Journal of High Performance Computing Applications*, 2011.

[2] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008.

[3] S. Seo, A. Amer, P. Balaji, P. Beckman *et al.*, "Argobots: A lightweight low-level threading/tasking framework," https://collab.cels.anl.gov/display/ARGOBOTS/.

[4] S. Pronk, S. Pall, R. Schulz, P. Larsson *et al.*, "GROMACS 4.5: A high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, 2013.

[5] M. Krone, J. E. Stone, T. Ertl, and K. Schulten, "Fast visualization of gaussian density surfaces for molecular dynamics and particle system trajectories," in *EuroVis Short Papers*, 2012.

[6] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CLUSTER)*, 2014.

[7] "Docker – build, ship, and run any app, anywhere," https://www.docker.com/.

[8] "APPC app container specification," https://github.com/appc/spec.

[9] "Linux control groups," https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[10] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, "DI-MMAP: A scalable memory map runtime for out-of-core data-intensive applications," *Cluster Computing*, 2015.

[11] M. Jiang, B. Van Essen, C. Harrison, and M. Gokhale, "Multi-threaded streamline tracing for data-intensive architectures," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2014.

[12] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT)*, 2004.

[13] A. Morari, R. Gioiosa, R. W. Wisniewski *et al.*, "A quantitative analysis of OS noise," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[14] ——, "Evaluating the impact of TLB misses on future HPC systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[15] S. Rostedt, "Finding origins of latencies using ftrace," in *Real Time Linux Workshop (RTLWS)*, 2009.

[16] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat, "SUNMOS for the Intel Paragon—A brief user's guide," in *Intel Supercomputing Users Group Meeting*, 1994.

[17] R. Riesen, R. Brightwell, P. G. Bridges *et al.*, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, 2009.

[18] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Resource management in the Tessellation manycore OS," in *USENIX Conference on Hot Topics in Parallelism, (HotPAR)*, 2010.

[19] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new OS abstractions," in *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[20] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, "Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK," in *ACM/IEEE Conference on Supercomputing (SC)*, 2010.

[21] Y. Park, E. Van Hensbergen, M. Hillenbrand *et al.*, "FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment," in *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012.

[22] B. Gerofi, A. Shimada, A. Hori, and Y. Ishikawa, "Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[23] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An architecture for extreme-scale operating systems," in *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2014.

[24] E. Shmueli, G. Almási, J. Brunheroto, J. Castaños, G. Dózsa, S. Kumar, and D. Lieber, "Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L," in *ACM International Conference on Supercomputing (ICS)*, 2008.

[25] K. Yoshii, K. Iskra, H. Naik, P. Beckman, and P. C. Broekema, "Performance and scalability evaluation of "Big Memory" on Blue Gene Linux," *International Journal of High Performance Computing Applications*, 2011.

[26] B. Kocoloski and J. Lange, "HPMMAP: Lightweight memory management for commodity operating systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[27] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "Operating system issues for petascale systems," *ACM SIGOPS Operating Systems Review*, 2006.

[28] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman, "The ghost in the machine: Observing the effects of kernel operation on parallel application performance," in *ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[29] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *ACM/IEEE Conference on Supercomputing (SC)*, 2008.

[30] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *ACM/IEEE Conference on Supercomputing (SC)*, 2003.

[31] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, 2008.

[32] R. Gioiosa, S. A. McKee, and M. Valero, "Designing OS for HPC applications: Scheduling," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2010.

[33] D. Wallace, "Compute Node Linux: Overview, progress to date & roadmap," in *Cray User Group Annual Technical Conference*, 2007.

[34] J. Lange, K. Pedretti, T. Hudson, P. Dinda *et al.*, "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010.

[35] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2013.

[36] D. Beserra, E. D. Moreno, P. T. Endo, J. Barreto, D. Sadok, and S. Fernandes, "Performance analysis of LXC for HPC environments," in *International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, 2015.

[37] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing Docker for HPC," in *Cray User Group Annual Technical Conference*, 2015.

[38] "rkt – a security-minded, standards-based container engine," https://coreos.com/rkt/.

[39] H. Yu, C. Wang, R. Grout, J. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, 2010.

[40] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, 2011.

[41] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2012.

[42] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky, "Just in Time: Adding value to the IO pipelines of high performance applications with JITStaging," in *International Symposium on High Performance Distributed Computing (HPDC)*, 2011.

[43] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf *et al.*, "FlexIO: I/O middleware for location-flexible scientific data analytics," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2013.