2016 IEEE International Parallel and Distributed Processing Symposium

On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel

Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Gou Nakamura[†], Tomoki Shirasawa[‡] and Yutaka Ishikawa
RIKEN Advanced Institute for Computational Science, JAPAN
[†]*Hitachi Solutions, Ltd., JAPAN*
[‡]*Hitachi Solutions East Japan, Ltd., JAPAN*
bgerofi@riken.jp, masamichi.takagi@riken.jp, ahori@riken.jp, go.nakamura.yw@hitachi-solutions.com,
tomoki.shirasawa.kk@hitachi-solutions.com, yutaka.ishikawa@riken.jp

***Abstract*—Extreme degree of parallelism in high-end computing requires low operating system noise so that large scale, bulk-synchronous parallel applications can be run efficiently. Noiseless execution has been historically achieved by deploying lightweight kernels (LWK), which, on the other hand, can provide only a restricted set of the POSIX API in exchange for scalability. However, the increasing prevalence of more complex application constructs, such as in-situ analysis and workflow composition, dictates the need for the rich programming APIs of POSIX/Linux. In order to comply with these seemingly contradictory requirements, hybrid kernels, where Linux and a lightweight kernel (LWK) are run side-by-side on compute nodes, have been recently recognized as a promising approach. Although multiple research projects are now pursuing this direction, the questions of how node resources are shared between the two types of kernels, how exactly the two kernels interact with each other and to what extent they are integrated, remain subjects of ongoing debate.**

In this paper, we describe IHK/McKernel, a hybrid software stack that seamlessly blends an LWK with Linux by selectively offloading system services from the lightweight kernel to Linux. Specifically, we are focusing on transparent reuse of Linux device drivers and detail the design of our framework that enables the LWK to naturally leverage the Linux driver codebase without sacrificing scalability or the POSIX API. Through rigorous evaluation on a medium size cluster we demonstrate how McKernel provides consistent, isolated performance for simulations even in face of competing, in-situ workloads.

***Keywords*-operating systems; hybrid kernels; lightweight kernels; system call offloading; scalability**

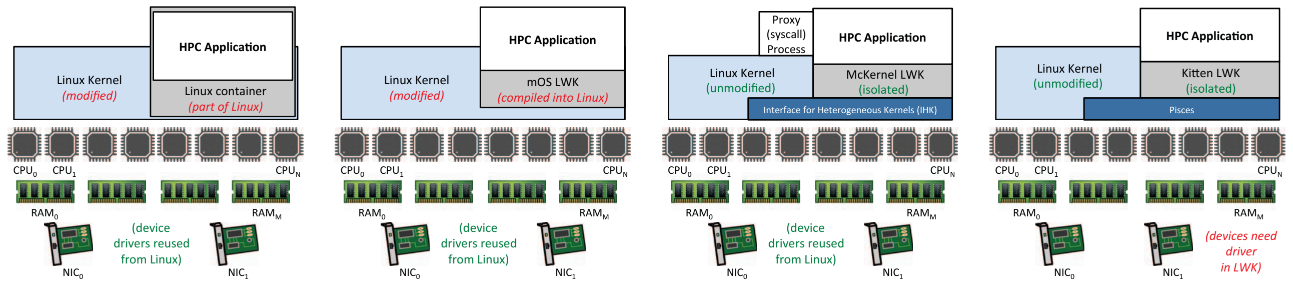
I. Introduction

With the growing complexity of high-end supercomputers, it has become indisputable that the current system software stack will face significant challenges as we look forward to exascale and beyond. The necessity to deal with extreme degree of parallelism, heterogeneous architectures, multiple levels of memory hierarchy, power constraints, etc. advocates operating systems that can rapidly adapt to new hardware requirements, and that can support novel programming paradigms and runtime systems. On the other hand, a new class of more dynamic and complex applications are also on the horizon, with an increasing demand for application constructs such as in-situ analysis, workflows, elaborate monitoring and performance tools [1], [2]. This complexity relies not only on rich features of POSIX, but also on the Linux APIs (such as the `/proc`, `/sys` filesystems, etc.) in particular.

Traditionally, lightweight operating systems specialized for HPC followed two approaches to tackle the high degree of parallelism so that scalable performance for bulk synchronous applications can be delivered. In the full weight kernel (FWK) approach [3], [4], [5], a full Linux environment is taken as the basis, and features that inhibit attaining HPC scalability are removed, i.e., making it lightweight. The pure lightweight kernel (LWK) approach [6], [7], [8], on the other hand, starts from scratch and effort is undertaken to add sufficient functionality so that it provides a familiar API, typically something close to that of a general purpose OS, while at the same time it retains the desired scalability and reliability attributes. Neither of these approaches yields a fully Linux compatible environment.

An alternative hybrid approach recognized recently by the system software community is to run Linux simultaneously with a lightweight kernel on compute nodes and multiple research projects are now pursuing this direction [9], [10], [11], [12]. The basic idea is that simulations run on an HPC tailored lightweight kernel, ensuring the necessary isolation for noiseless execution of parallel applications, but Linux is leveraged so that the full POSIX API is supported. Additionally, the small code base of the LWK can also facilitate rapid prototyping for new, exotic hardware features [13], [14], [15]. Nevertheless, the questions of how to share node resources between the two types of kernels, where do device drivers execute, how exactly do the two kernels interact with each other and to what extent are they integrated, remain subjects of ongoing debate.

Figure 1 illustrates the hybrid/specialized LWK landscape highlighting kernel level workload isolation, reusability of Linux device drivers, and necessary Linux kernel modifications. It is worth emphasizing that modifications to the Linux kernel are highly undesired since Linux is a rapidly evolving target and keeping patches up-to-date with the latest kernel can pose a major challenge. Generally, the left side of the figure represents tight integration between Linux and the LWK, while progressing to the right gradually enforces



(a) Argo, led by Argonne National Labs. (b) mOS @ Intel Corporation. (c) IHK/McKernel @ RIKEN. (d) Hobbes, led by Sandia National Labs.

Figure 1. **Overview of the Hybrid/LWK landscape from the perspectives of required Linux kernel modifications, kernel level workload isolation and device driver reusability.** *Bracketed green labels indicate advantages, while bracketed red labels (in italic) indicate disadvantages.*

stricter isolation[1] The Argo project, shown in Figure 1a, is investigating the feasibility of Linux containers for large scale HPC by means of extending the Linux kernel so that the necessary resource isolation can be attained [16]. While it comes at the price of modifications to Linux, from a low-level driver point of view their specialized HPC containers can directly leverage Linux managed devices. Nonetheless, the biggest concern with Argo's approach is the uncertainty whether or not jitter free execution can be sufficiently achieved.

mOS [9] (Figure 1b), a hybrid kernel approach currently pursued by Intel, represents a significant departure from the all Linux based solution of Argo. mOS provides its own LWK, which runs on a dedicated partition of hardware resources, but it compiles the LWK codebase into Linux aiming at exploiting the Linux infrastructure as much as possible. mOS also keeps the lightweight kernel internals Linux compatible on the level of various kernel data structures so that it can migrate LWK threads into Linux for system call execution [17]. While this approach also enables access to Linux device drivers, mOS occasionally runs Linux code on the LWK cores and again, its ability to acquire a fully jitterless execution environment remains to be seen.

On the right end of the spectrum is the Hobbes stack [12] (i.e., Sandia's Kitten lightweight kernel over the Pisces resource manager) running the LWK in its completely isolated resource partition. The principal idea of this configuration is to enforce full isolation between Linux and the LWK and thus to avoid interference between the two kernels entirely. In fact, Hobbes is the only hybrid kernel so far which has demonstrated (on single node experiments) that it truly has the potential to guarantee noiseless execution of HPC simulations in face of in-situ workloads [18]. Unfortunately, device drivers in Hobbes need to be ported to the LWK for every single device the kernel needs to handle, and the lack of high performance network drivers has been the hindering obstacle for performing a larger scale evaluation.

In this paper, we present IHK/McKernel, illustrated in Figure 1c, which we designed in a way so that it provides sufficient kernel level isolation and transparent access to Linux device drivers at the same time. Furthermore, it requires no modifications to Linux. We run our lightweight kernel (i.e., McKernel) on its dedicated partition of CPU cores and physical memory, but via selectively offloading OS services to Linux we can seamlessly leverage the Linux device driver codebase. We summarize our contributions as follows:

- We introduce a hybrid Linux plus LWK kernel organization that provides transparent access to Linux device drivers via selectively offloading OS services without sacrificing LWK scalability or the support for full POSIX/Linux APIs;
- Specifically, we propose the concept of *unified address space* and detail the mechanism of device file mappings which both are key factors to seamless device driver reuse;
- As opposed to previous claims [18], we demonstrate that offloading certain system calls does not obstruct scalable performance in a hybrid kernel setting;
- And, to the best of our knowledge for the first time, we provide a rigorous evaluation of a hybrid HPC kernel at scale.

We find that IHK/McKernel does not only succeed in containing the OS noise of Linux when running in-situ tasks (providing up to an order of magnitude less performance variation of certain MPI collective operations than that on Linux), but it also outperforms Linux on various applications even without competing workloads.

The rest of this paper is organized as follows. We begin with providing background information in Section II. Section III discusses design issues of address space sharing and transparent device driver support. Experimental evaluation is given in Section IV. Section V surveys related work, and finally, Section VI concludes the paper.

[1]Note that we intentionally describe some of the closely related projects here to further motivate our approach, but a more general overview of related work will be presented in Section V.

II. Background

In order to provide the basis for discussion on device drivers, we first present background information on the IHK/McKernel hybrid stack. An architectural overview of the main system components is shown in Figure 2.

At the heart of the stack is a low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [11]. IHK is a general framework that provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. Note that IHK can allocate and release host resources dynamically and no reboot of the host machine is required when altering configuration. The latest version of IHK is implemented as a collection of kernel modules without any modifications to the kernel code itself, which enables straightforward deployment on a wide range of Linux distributions. Besides resource and LWK management, IHK also provides an Inter-Kernel Communication (IKC) layer, upon which system call delegation is implemented (see Section III for further details).

McKernel is a lightweight kernel written from scratch. It is designed for HPC and it is booted from IHK. McKernel retains a binary compatible ABI with Linux, however, it implements only a small set of performance sensitive system calls and the rest are delegated to Linux. Specifically, McKernel has its own memory management, it supports processes and multi-threading with a simple round-robin cooperative (tick-less) scheduler, and it implements signaling. It also allows inter-process memory mappings and it provides interfaces to hardware performance counters.

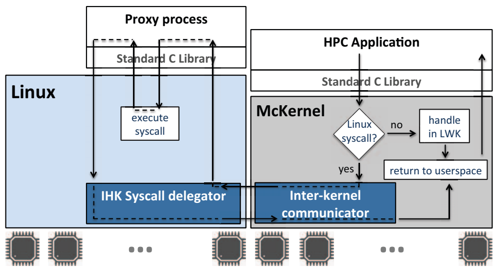


Figure 2. **Overview of the IHK/McKernel architecture and the system call delegation mechanism.**

For each process running on McKernel there is a process created on the Linux side, which we call the *proxy-process*. The proxy process' central role is to facilitate system call offloading. Essentially, it provides execution context on behalf of the application so that offloaded calls can be directly invoked in Linux. The proxy process also enables Linux to maintain certain state information that would have to be otherwise kept track of in the LWK. McKernel for instance has no notion of file descriptors, but rather it simply returns the number it receives from the proxy process when a file is opened. The actual set of open files (i.e., file descriptor table, file positions, etc..) are managed by the Linux kernel. We emphasize that IHK/McKernel runs HPC applications primarily on the LWK to achieve noiseless execution but the full Linux API is available via system call delegation.

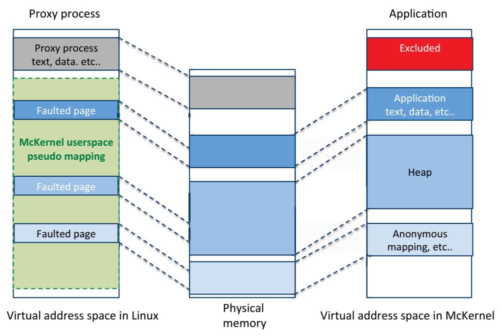
III. Design and Implementation

This section details two important aspects of the Linux and McKernel symbiosis. First we introduce the notion of *unified address space* between the application and its corresponding proxy process. We then discuss how device drivers are made accessible to McKernel.

A. Unified Address Space

To motivate the need for unified address space, we begin with a more detailed description of the system call offloading mechanism, which is illustrated in Figure 2. During system call delegation McKernel marshalls the system call number along with its arguments and sends a message to Linux via a dedicated IKC channel. The corresponding proxy process running on Linux is by default waiting for system call requests through an `ioctl()` call into IHK's system call delegator kernel module. The delegator kernel module's IKC interrupt handler wakes up the proxy process, which returns to userspace and simply invokes the requested system call. Once it obtains the return value, it instructs the delegator module to send the result back to McKernel, which subsequently passes the value to user-space.

Notice, however, that certain system call arguments may be merely pointers (e.g., the buffer argument of a `read()` system call) and the actual operation takes place on the contents of the referred memory. Thus, the main problem is how the proxy process on Linux can resolve virtual addresses in arguments so that it can access the memory of the application running on McKernel.



Proxy process
Application
Proxy process text, data, etc..
Excluded
Faulted page
Application text, data, etc..
McKernel userspace pseudo mapping
Heap
Faulted page
Faulted page
Anonymous mapping, etc..
Virtual address space in Linux
Physical memory
Virtual address space in McKernel

Figure 3. **Conceptual overview of the *unified address space* between the application and its corresponding proxy process.**

The unified address space model in IHK/McKernel ensures that offloaded system calls can seamlessly resolve arguments even in case of pointers. This mechanism is depicted in Figure 3 and it is implemented as follows. First, the proxy process is compiled as a position independent binary, which enables us to map the code and data segments specific to the proxy process to an address range which is explicitly excluded from McKernel's user space. The red box on the right side of the figure demonstrates the excluded region. Second, the entire valid virtual address range of McKernel's application user-space is covered by a special mapping in the proxy process for which we use a pseudo file mapping in Linux. This mapping is indicated by the green, dashed box on the left side of the figure.

Note, that the proxy process does not need to fill in any virtual to physical mappings at the time of creating the pseudo mapping and it remains empty unless an address is referenced. Every time an unmapped address is accessed, however, the page fault handler of the pseudo mapping consults the page tables corresponding to the application on the LWK and maps it to the exact same physical page. Such mappings are demonstrated in the figure by the small boxes on the left labeled as *faulted page*. This mechanism ensures that the proxy process, while executing system calls, has access to the same memory content as the application. Needless to say, Linux' page table entries in the pseudo mapping have to be occasionally synchronized with McKernel, for instance, when the application calls `munmap()` or modifies certain mappings.

B. Device Driver Transparency

Applications running on UNIX like operating systems normally interact with devices either through I/O system calls (e.g., `read()`, `write()`, `ioctl()`, etc.) on dedicated device files or by mapping device memory directly into user-space.

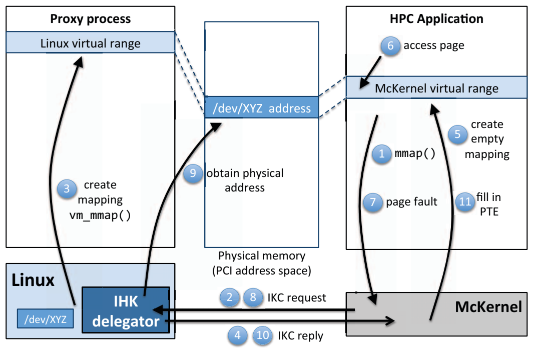


Figure 4. **Mapping device files in McKernel.**

We have already mentioned that McKernel does not implement I/O calls, but instead, it forwards them to Linux. Thus, with the help of unified address space applications running on McKernel can transparently interact with devices using I/O system calls. What is more challenging in a hybrid kernel setting, however, is to provide support for memory mapped device files. We have carefully designed IHK and McKernel in a way so that devices can be memory mapped without any modification to existing driver code.

Figure 4 demonstrates the main steps of mapping device files in McKernel, which can be summarized as follows. ① The application invokes the `mmap()` system call on a device file and ② McKernel forwards the request to the delegator IHK component in Linux. ③ The kernel module memory maps the device file into the proxy process' address space and creates a tracking object that will be used to serve future page faults. ④ Linux replies to McKernel so that ⑤ it can also allocate its own virtual memory range in the address space of the application. Note that in the proxy process (on the Linux side) the entire valid user-space of the actual application is covered by the unified address space's pseudo mapping and thus the two mappings result in different virtual addresses.

The most important observation, however, is that although the virtual memory ranges in Linux and in McKernel are different, the proxy process on Linux will never access its mapping, because the proxy process never runs actual application code. Rather, the following steps occur. ⑥ The application accesses an address in the mapping; and ⑦ causes a page fault. ⑧ McKernel's page fault handler knows that the device mapping requires special attention and it requests the IHK module on Linux to ⑨ resolve the physical address based on the tracking object and the offset in the mapping. ⑩ Linux replies the request and ⑪ McKernel fills in the missing page table entry.

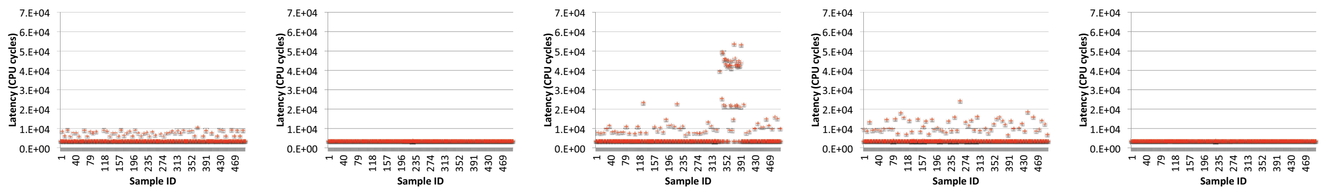
It is worth pointing out that in modern high performance networks (such as Infiniband [19]) device mappings are usually established in application initialization phase and the actual interaction with the device is comprised of mostly regular `load/store` instructions carried out entirely in user-space.

IV. Evaluation

A. Experimental Environment

All experiments were performed on a middle size cluster comprised of 64 compute nodes. Each node is equipped with a two sockets 2.8GHz Intel® Xeon® CPU E5-2680 (10 cores per socket) and 64GB of RAM arranged in two NUMA domains (i.e., ten CPU cores each). Additionally, there are two types of interconnection networks available on each compute node, Gigabit Ethernet and Mellanox Technologies MT27600 Connect-IB FDR 56 Gb/s.

The software environment we used is as follows. Compute nodes run Red Hat Enterprise Linux ComputeNode Release 6.5 with Linux kernel version 2.6.32. For McKernel measurements we utilized IHK's dynamic partitioning feature to reserve CPU cores and physical memory. In all experiments, we used the MVAPICH 2.2a MPI distribution with Intel Compiler version 16.0.0. For measurements that involve in-situ workloads, we ran Hadoop version 2.7.1 with some of the builtin benchmarks as well as the pagerank workload from the HiBench benchmark suite [20]. We emphasize that *there is absolutely no need to modify applications* in order to



(a) Linux+cgroup (b) McKernel (c) Linux+cgroup with Hadoop (d) Linux+cgroup+isolcpus with Hadoop (e) McKernel with Hadoop

Figure 5. **FWQ noise measurements for Linux and McKernel with and without competing Hadoop workload.**

run them on McKernel and we used the exact same binaries for measurements running on top of Linux and our stack. Furthermore, we have done *no modifications to the MPI library* either and access to the Infiniband network is assured by McKernel's transparent device driver support.

We were curious to assess mainly two aspects of IHK/McKernel. First, we compare the scalability of various micro-benchmarks and mini applications depending on the underlying operating system (Linux versus McKernel), both in terms of absolute runtime as well as in performance variation across multiple runs. Second, we seek to characterize McKernel's ability to provide consistent performance of HPC simulations in face of in-situ, competing workloads. Co-locating data analytics with HPC has been reportedly gaining importance [2] and this study focuses on the ability of the operating system to avoid interference by the analytics workload and to ensure noiseless execution environment for HPC simulations. In addition, part of our experiments is demonstrating to what extent Linux can succeed in isolating operating system noise depending on various configuration options. It is worth emphasizing that we do not focus on the in-situ workload itself and we simply deploy representative data analytics applications. Investigating, for instance, how far Hadoop performance is affected by co-located simulations falls outside the scope of this paper. Furthermore, we do not explicitly deal with communication between the simulation and in-situ processes and simply assume that a straightforward shared memory segment would be sufficient.

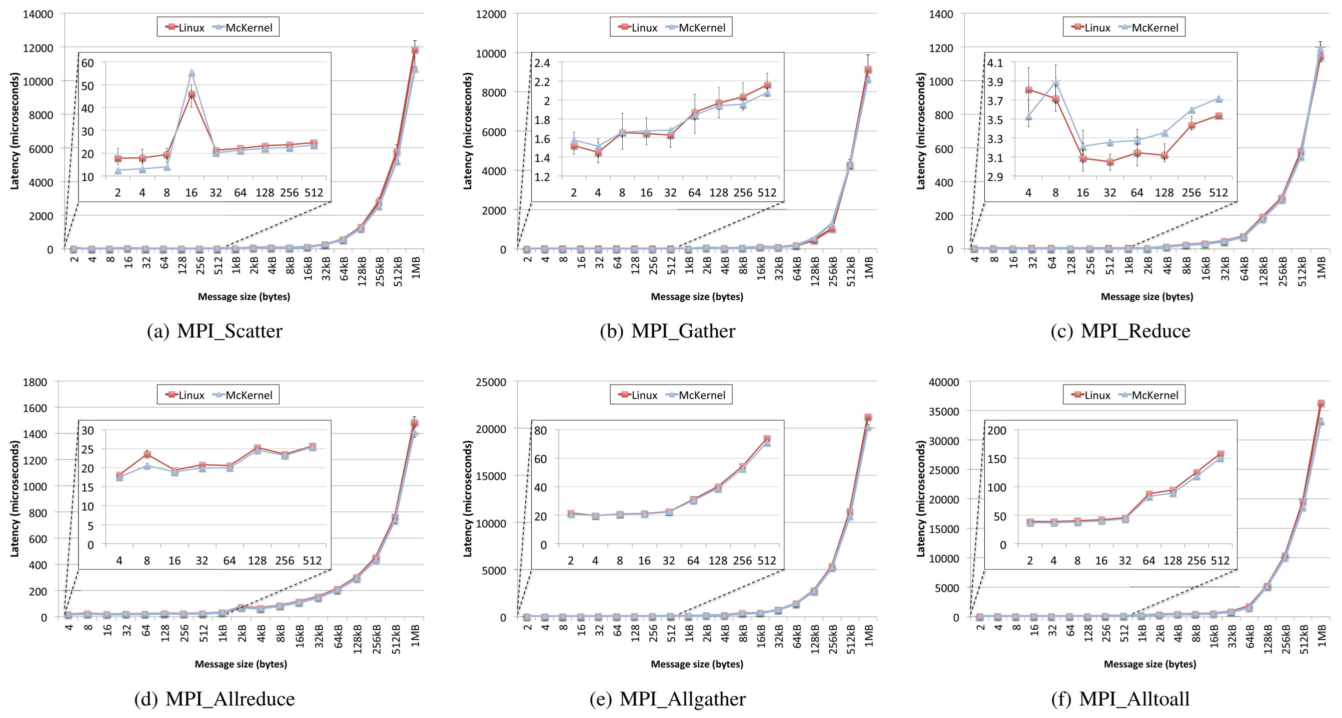
In order to provide fair comparison, in all experiments that contrast Linux and McKernel it is ensured that applications run on the exact same set of CPU cores and use the same NUMA memory domain for both operating systems. Specifically, HPC simulations are always bound to the physical memory and CPU cores of NUMA node one, running either on Linux or McKernel, while Hadoop runs on Linux bound to NUMA node zero (unless stated otherwise). When running IHK/McKernel, this arrangement translates to running McKernel on top of 9 CPU cores in NUMA node one and assigning the remaining single core to the proxy process. It is also worth pointing out that Infiniband is exclusively used by HPC simulations and Hadoop utilizes Gigabit Ethernet so that network traffic between the two types of workloads is completely isolated.

B. Results

1) Single Node OS Noise: The first set of experiments we performed was to compare OS noise of Linux and McKernel both with and without in-situ workloads. We used the Fixed Work Quantum (FWQ) test from the ASC Sequoia Benchmark Codes [21]. The FWQ benchmark measures hardware and software interference by repetitively performing a fixed amount of work (the work quanta), measuring the time necessary to complete the task.

We measured multiple 30 seconds intervals and report the values where OS noise was the most significant, because we are interested in the worst case scenario. Figure 5 indicates the results. We ran five configurations. In the measurements without in-situ workloads we simply compare RedHat Linux and McKernel (shown in Figure 5a and Figure 5b, respectively). While in case of McKernel IHK explicitly reserves CPU cores and memory for the LWK, for Linux we used the `cgroup` facility to ensure the tests run under the same hardware conditions. As seen, although Linux provides a fairly low OS noise when idle, McKernel's values are virtually constant. This was expected, because McKernel runs no device drivers, it is tick-less and features only a co-operative scheduler, which together eliminate any undesired asynchronous kernel events.

What is more interesting, however, is to assess the effect of an in-situ, competing workload. We co-locate Hadoop in this test and compare two Linux configurations with McKernel. The *Linux+cgroup* setup indicates that FWQ is pinned to specific CPU cores, but there is no restriction on where Hadoop processes execute. On the other hand, for the *Linux+cgroup+isolcpus* scenario we use the Linux kernel's `isolcpus` kernel argument to exclude specific CPU cores from the core Linux scheduler. FWQ is then explicitly run on those cores. The expected effect of this configuration is much better performance isolation, because Linux is prohibited to schedule any processes on the designated cores (unless it is explicitly instructed to do so via `cgroup` or `taskset`, etc.). As seen, the *Linux+cgroup* configuration (Figure 5c) is most effected by the Hadoop workload experiencing up to 16X slowdown compared to the expected value. *Linux+cgroup+isolcpus* (Figure 5d) does improve the situation by a good margin, but it still shows significant variation. On the other hand, as shown in Figure



(a) MPI_Scatter (b) MPI_Gather (c) MPI_Reduce

(d) MPI_Allreduce (e) MPI_Allgather (f) MPI_Alltoall

Figure 6. **OSU MPI benchmark results for various collective operations. Measurements compare Linux with McKernel using 64 compute nodes.**

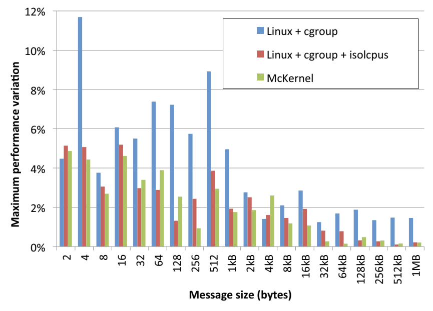
5e, McKernel experiences no disturbance at all. Again, this was highly expected since IHK/McKernel separates the two workloads at the OS kernel level.

2) MPI Collective Communication: The next set of measurements focus on communication scalability and the impact of in-situ workloads on communication performance. To assess message passing scalability when running on top of McKernel we measured various collective operations and compared their performance to Linux. We emphasize again that our main motivation for providing these measurements is to demonstrate IHK/McKernel's ability to enable transparent access to the Infiniband network in a hybrid kernel setting without any modifications to device drivers or to the MPI library.

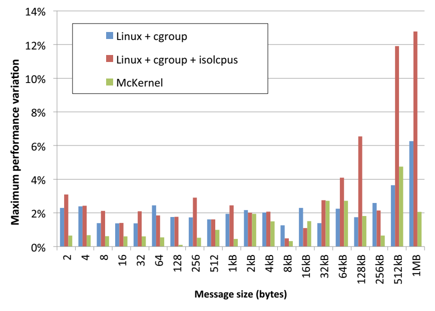
Specifically, we used the OSU benchmark suite from the MVAPICH 2.2a distribution [22] and measured the performance of collective operations using 64 nodes. Figure 6 indicates messaging latency as the function of message size. We run each experiment 15 times and report the average value with the error bars representing performance variation. Generally, McKernel yields similar performance to Linux with occasional differences. For instance, except for a couple of cases we observe slightly better performance for `MPI_Scatter` and `MPI_Gather` running on McKernel, shown in Figure 6a and Figure 6b, respectively. On the other hand, as Figure 6c reveals, `MPI_Reduce` yields somewhat better values when running on Linux especially for small messages. Measurements for `MPI_Allreduce`, `MPI_Allgather` and `MPI_Alltoall` are indicated by Figures 6d, 6e and 6f, respectively. As earlier, both OS yield similar performance results for most message sizes, for some messages McKernel slightly outperforming Linux. Nevertheless, the most important observation across all measurements is the visibly lower performance variation of McKernel, which indicates smaller OS jitter and thus a more consistent execution.

Furthermore, the difference between performance variations over Linux and McKernel becomes much more pronounced when introducing in-situ workloads. Hadoop was configured to involve all 64 nodes in computation and we used the same Linux configurations for isolating applications as described previously.

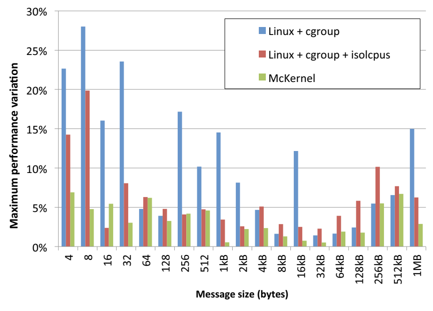
Because we did not observe significant changes in average performance compared to the baseline experiments, we focus plainly on variation. Figure 7 depicts the results, where the Y axis indicates the maximum variation in percentage compared to the average value. As seen, in average across all measurements, McKernel provides substantially lower variation than Linux, yielding approximately $2-6\%$ versus the Linux values that go up to 29%. Nevertheless, there are cases where McKernel gets close or even becomes slightly worse than the *Linux+cgroup+isolcpus* setup, especially for large messages in case of `MPI_Reduce` and `MPI_Allreduce`, see Figure 7c and Figure 7e, respectively. We have investigated the root cause for this phenomena and found that for large message sizes the MPI implementation often utilizes internal buffers which need to be registered for Infiniband's RDMA engine. Because the registration



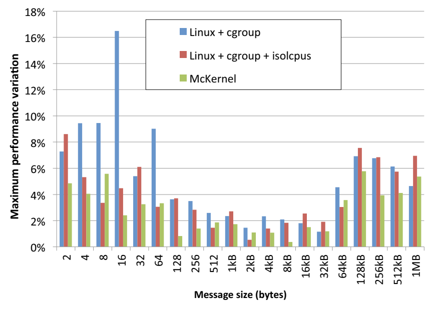
(a) MPI_Scatter



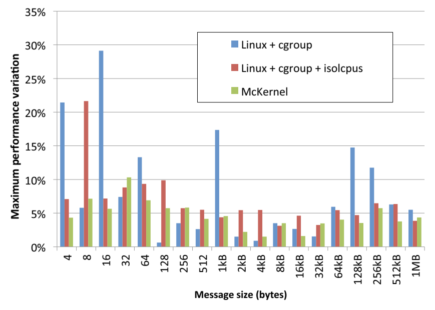
(b) MPI_Gather



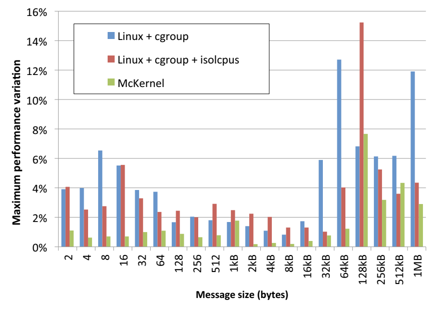
(c) MPI_Reduce



(d) MPI_Allgather



(e) MPI_Allreduce



(f) MPI_Alltoall

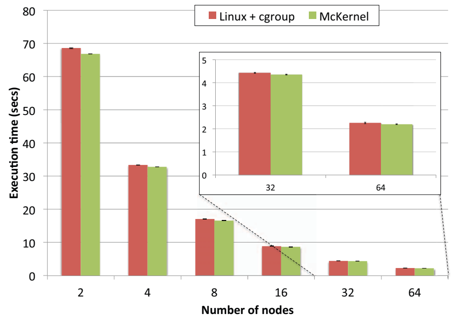
Figure 7. **Performance variation of OSU MPI benchmarks collective operations co-located with competing Hadoop workload. Measurements compare various Linux configurations with McKernel using 64 compute nodes.**

operation is performed through a `write()` system call, it gets offloaded even in case of McKernel. This is currently an issue in our implementation, but making MPI aware of the hybrid setting could easily solve the problem. Additionally, it is also worth noting that certain hardware components (e.g., the last level cache) are shared, which we cannot control in software. Another surprising result is the occasionally higher variation of the *Linux+cgroup+isolcpus* setup compared to *Linux+cgroup* (see Figure 7b), which suggests that even using `isolcpus`, OS noise on Linux can be significant.

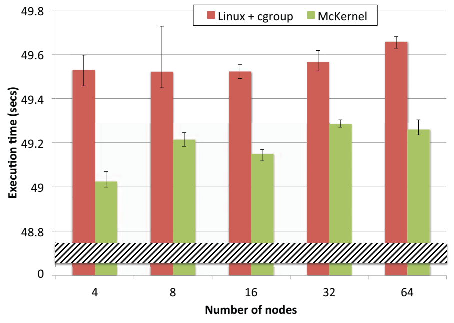
3) Mini Applications: In the next set of experiments we turn our attention to application level performance. We deployed the miniFE and HPC-CG miniapplications from Sandia's Mantevo suite [23] plus used Modylas and FFVC from RIKEN's Fiber miniapplications package [24]. These applications cover a relatively wide set of domains ranging from finite element computation (miniFE) through a more focused sparse iterative solver (HPC-CG) via a molecular dynamics simulation (Modylas) to a fluid dynamics code (FFVC). All applications utilize MPI+OpenMP and we set the number of threads per node to 8 (the largest number which is power of two and still fits into one NUMA domain on our platform). Note that miniFE and Modylas are strong scaling, while HPC-CG and FFVC are weak scaling applications.

The first measurements compare plain execution of applications, depicted by Figure 8. Again, we report average execution time with error bars indicating variation. Much to our surprise, McKernel slightly outperforms Linux for most of the workloads yielding between $1-8\%$ improvement. Furthermore, one can also notice lower performance variation, which is especially true for HPC-CG, shown in Figure 8b. Performance counters revealed that McKernel yields in average 1% and 3% less TLB and LLC misses, respectively, which we suspect is the result of contiguous physical memory behind anonymous mappings. Additionally, it is our policy to have McKernel reinitialized between subsequent executions so that each time the applications can start from a clean and consistent kernel state.

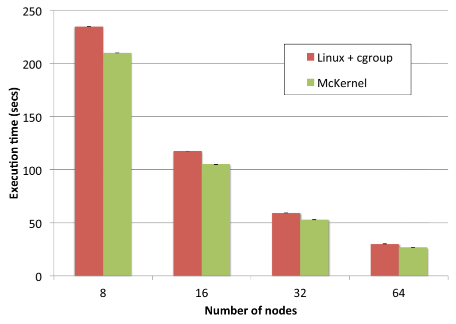
The last set of experiments evaluates the IHK/McKernel stack's ability to prevent interference between HPC simulations and co-located in-situ workloads. As mentioned earlier, we ran HPC on NUMA domain one and various Hadoop workloads on NUMA node zero. Figure 9 indicates the results. In general, we found that HPC-CG and FFVC, see Figure 9b and Figure 9d, respectively, were the most severely affected by OS noise in particular when running under the *Linux+cgroup* configuration. While *Linux+cgroup+isolcpus* visibly decreases performance variation it fails to attain the same level of workload isolation as McKernel. Thus, the most important observation is that McKernel provides substantially lower performance variation across all applications compared to the Linux runs. Specifically, the highest variation across all workloads was $3.1X$ and 16% for the *Linux+cgroup*, the *Linux+cgroup+isolcpus* configurations, respectively, while McKernel yielded only 3% difference in



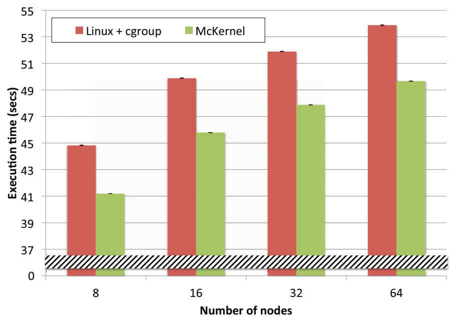
(a) miniFE



(b) HPC-CG



(c) Modylas



(d) FFVC

Figure 8. **Various MiniApp performance comparing Linux and McKernel.** *Note that for clarity, the right side figures' Y axis are clipped.*

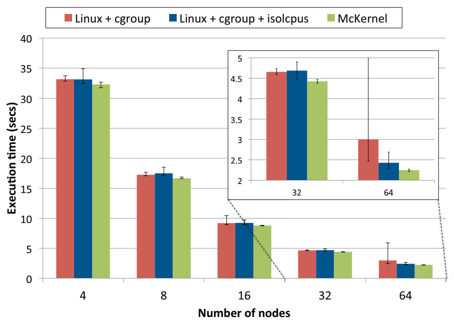
the worst case. It is also worth noting that although 64 nodes is still a relatively small scale compared to current high-end systems, moving to larger scales would further exacerbate the issue of interference.

V. Related Work

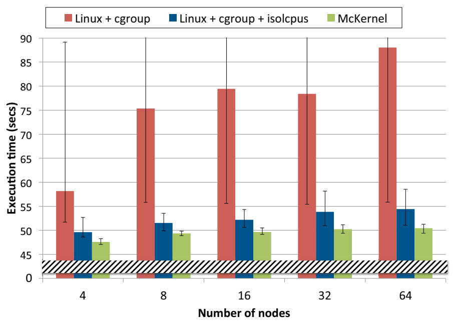
Lightweight Kernels: Lightweight kernels specifically designed for HPC workloads date back over 20 years now. Catamount [6] from Sandia National Laboratories was one of the notable systems which has been developed from scratch and successfully deployed on a large scale supercomputer. The IBM BlueGene line of supercomputers have also been running an HPC targeted lightweight kernel called CNK [7]. Nevertheless, CNK borrows a significant amount of code from Linux (e.g., glibc, NPTL) so that it can comply with elaborate Unix features, which have been increasingly demanded by the growing complexity of nowadays' HPC applications. The most current in Sandia National Lab's lightweight compute node kernels line of effort is Kitten [8], which distinguishes itself from their prior LWKs by providing a more complete Linux-compatible user environment. It also features a virtual machine monitor capability via Palacios [25] that allows full-featured guest OSs. However, with the ever growing appetite for full Unix/POSIX feature compatibility from the application side, it has become increasingly difficult to support all these features without compromising the primary goal of LWK performance.

On the other end of the lightweight kernel spectrum are kernels which originate from Linux, but have been heavily modified to meet HPC requirements ensuring low noise, scalability and predictable application performance. Cray's Extreme Scale Linux [4] and ZeptoOS [5] follow this approach. They often employ techniques, such as eliminating daemon processes, simplifying the scheduler or replacing the memory management system. There are mainly two problems with the Linux approach. First, the heavy modifications occasionally break Linux compatibility, which is highly undesired. Second, because HPC tends to follow rapid hardware changes that need to be reflected in kernel code, Linux often falls behind with the necessary updates which results in an endless need for maintaining Linux patches.

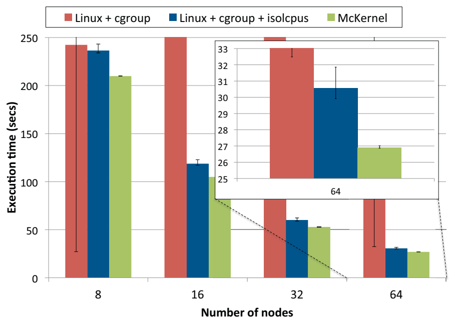
Hybrid Kernels for HPC: FusedOS [26] was the first proposal to combine Linux with an LWK. It's primary objective was addressing core heterogeneity between system and application cores and at the same time providing a standard operating environment. Contrary to McKernel, FusedOS runs the LWK at user level. In the FusedOS prototype, the kernel code on the application core is simply a stub that offloads all system calls to a corresponding user-level proxy process called CL. The proxy process itself is similar to that in IHK/McKernel, but in FusedOS the entire LWK is implemented within the CL process on Linux. The FusedOS work was the first to demonstrate that Linux noise can be isolated to the Linux cores and avoid interference with the



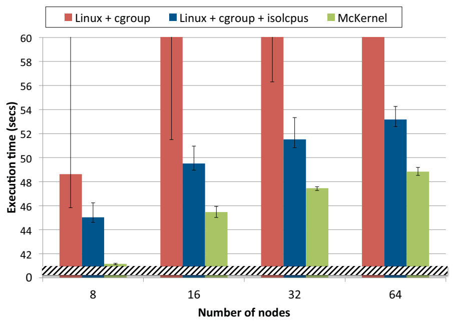
(a) miniFE



(b) HPC-CG



(c) Modylas



(d) FFVC

Figure 9. **The effect of competing Hadoop workload on various MiniApp performance.** *Note that for clarity, the right side figures' Y axis are clipped.*

HPC application running on the LWK cores. This property has been also one of the main driver for the McKernel model.

From more recent hybrid kernels, one of the most similar efforts to our work is Intel's mOS project [9], [17]. The most important difference between McKernel and mOS is the way how LWK and Linux are integrated. mOS takes a path of much stronger integration with the intention of minimizing LWK development and to directly take advantage of the Linux infrastructure. Nevertheless, this approach comes at the cost of Linux modifications and an increased complexity of eliminating OS noise.

Hobbes [12] is one of the DOE's ongoing Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to explicitly support application composition, which is emerging as a key approach for applications to address scalability and power concerns anticipated with coming extreme-scale architectures. Hobbes also makes use of virtualization technologies to provide the flexibility to support requirements of application components for different node-level operating systems and runtimes. At the bottom of the software stack, Hobbes relies on Kitten [8] as its LWK component, on top of which Palacios [25] is in charge to serve as a virtual machine monitor. As opposed to IHK/McKernel, Hobbes separates Linux and Kitten at the PCI device level, which imposes difficulties both for providing a full POSIX API and the necessary driver support in the LWK.

Argo [10] is another DOE OS/R project targeted at applications with complex work flows. Argo envisions using OS and runtime specialization (via enhanced Linux containers) inside compute nodes. In Argo's architecture, each node may contain a heterogeneous set of compute resources, a hierarchy of memory types with different performance (bandwidth, latency) and power characteristics. Given such a node architecture, Argo expects to use a ServiceOS like Linux to boot the node and run management services. It then expects to run different container instances that cater to the specific needs of applications.

Operating System Noise: OS noise has been shown to be a key limiter of application scalability in high-end systems. For example, Beckman et. al investigated the effect of OS jitter on collective operations and concluded that performance is often correlated to the largest interruption to the application [27]. Ferreira et. al used a kernel-based noise injection mechanism to characterize the effect of OS noise on application performance [28], while Hoefler et. al used a simulation to study the same subject [29]. Although these studies help to understand OS noise in a standalone setting, we are focusing on hybrid kernels and in particular, on the effect of in-situ workloads. A very recent study has investigated similar issues in the context of the Hobbes project [18], due to the lack of device drivers in the Kitten LWK, however, the authors could only provide single node measurements when using their lightweight kernel.

VI. Conclusion and Future Work

Hybrid kernel architectures have received a great deal of attention recently due to their potential for addressing many of the challenges system software faces as we move towards exascale and beyond. However, many questions regarding how multiple kernels interplay remain open.

This paper has presented IHK/McKernel, a hybrid Linux+LWK OS stack that provides LWK capabilities for noiseless execution of HPC simulations, retains the full POSIX/Linux APIs and enables transparent access to Linux device drivers. At the same time, it requires no modifications to the Linux kernel. McKernel outperforms Linux on a range of applications and it can successfully avoid interference between HPC simulations and competing in-situ workloads.

In the future, we will further investigate eliminating the RDMA registration issue and we also intend to evaluate IHK/McKernel on much larger scale.

Acknowledgment

This work has been partially funded by MEXT's program for the Development and Improvement of Next Generation Ultra High-Speed Computer Systems.

We acknowledge Toshihiro Hanawa and Kengo Nakajima from the University of Tokyo for letting us use the KNSC cluster and for their technical support and assistance.

References

[1] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform," in *Proceedings of IPDPS'12*, May 2012, pp. 1352–1363.

[2] D. Tiwari, S. Boboila, S. S. Vazhkudai, Y. Kim, X. Ma, P. J. Desnoyers, and Y. Solihin, "Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines," in *Proceedings of FAST'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 119–132.

[3] S. Oral, F. Wang, D. A. Dillow, R. Miller, G. M. Shipman, D. Maxwell, D. Henseler, J. Becklehimer, and J. Larkin, "Reducing Application Runtime Variability on Jaguar XT5," in *Proceedings of CUG'10*, 2010.

[4] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems," in *Proceedings of CUG'12*, 2012.

[5] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema, "Characterizing the Performance of Big Memory on Blue Gene Linux," in *Proceedings of ICPPW'09*. IEEE Computer Society, 2009, pp. 65–72.

[6] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Proceedings of CUG'05*, 2005, pp. 16–19.

[7] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK," in *Proceedings of SC '10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[8] "Kitten: A Lightweight Operating System for Ultrascale Supercomputers (Accessed: Sep, 2015)," https://software.sandia.gov/trac/kitten.

[9] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An Architecture for Extreme-scale Operating Systems," in *Proceedings of ROSS '14*. New York, NY, USA: ACM, 2014, pp. 2:1–2:8.

[10] "Argo: An Exascale Operating System (Accessed: Sep, 2015)," http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[11] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa, "Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures," in *Proceedings of HiPC '14*, Dec 2014, pp. 1–10.

[12] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R," in *Proceedings of ROSS'13*. New York, NY, USA: ACM, 2013, pp. 2:1–2:8.

[13] 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 *Proceedings of CCGRID'13*, May 2013, pp. 360–368.

[14] Y. Soma, B. Gerofi, and Y. Ishikawa, "Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach," in *Proceedings of ROSS '14*. New York, NY, USA: ACM, 2014, pp. 3:1–3:8.

[15] B. Gerofi, A. Shimada, A. Hori, T. Masamichi, and Y. Ishikawa, "CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores," in *Proceedings of HPDC'14*. New York, NY, USA: ACM, 2014, pp. 73–84.

[16] J. A. Zounmevo, S. Perarnau, K. Iskra, K. Yoshii, R. Gioiosa, B. C. V. Essen, M. B. Gokhale, and E. A. Leon, "A Container-Based Approach to OS Specialization for Exascale Computing," in *Proceedings of WoC'15*, March 2015.

[17] B. Gerofi, M. Takagi, Y. Ishikawa, R. Riesen, E. Powers, and R. W. Wisniewski, "Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing," in *Proceedings of ROSS'15*. ACM, 2015, pp. 5:1–5:8.

[18] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti, "Achieving Performance Isolation with Lightweight Co-Kernels," in *Proceedings of HPDC'15*. New York, NY, USA: ACM, 2015, pp. 149–160.

[19] "InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2."

[20] "HiBench Suite: The bigdata micro benchmark suite." https://github.com/intel-hadoop/HiBench.

[21] "Fixed Time Quantum and Fixed Work Quantum Tests (Accessed: Sep, 2015)," https://asc.llnl.gov/sequoia/benchmarks.

[22] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE (Accessed: Sep, 2015)," http://mvapich.cse.ohio-state.edu/.

[23] "Mantevo Suite (Accessed: Sep, 2015)," https://mantevo.org/default.php.

[24] "Fiber MiniApp Suite (Accessed: Sep, 2015)," http://fiber-miniapp.github.io/.

[25] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing," in *Proceedings of IPDPS'10*, April 2010, pp. 1–12.

[26] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. Wisniewski, "FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment," in *Proceedings of SBAC-PAD'12*, Oct 2012, pp. 211–218.

[27] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale," in *Proceedings of Cluster'16*, Sept 2006, pp. 1–12.

[28] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection," in *Proceedings of SC'08*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 19:1–19:12.

[29] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *Proceedings of SC'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.