2018 IEEE International Parallel and Distributed Processing Symposium

Performance and Scalability of Lightweight Multi-Kernel based Operating Systems

Balazs Gerofi, Rolf Riesen[†], Masamichi Takagi, Taisuke Boku[‡], Kengo Nakajima[§],
Yutaka Ishikawa and Robert W. Wisniewski[†]
RIKEN Advanced Institute for Computational Science, Japan
[†]Intel Corporation, USA
[‡]University of Tsukuba, Japan
[§]University of Tokyo, Japan
bgerofi@riken.jp, rolf.riesen@intel.com, masamichi.takagi@riken.jp, taisuke@cs.tsukuba.ac.jp,
nakajima@cc.u-tokyo.ac.jp, yutaka.ishikawa@riken.jp, robert.w.wisniewski@intel.com

***Abstract*—Multi-kernels leverage today's multi-core chips to run multiple operating system (OS) kernels, typically a Light Weight Kernel (LWK) and a Linux kernel, simultaneously. The LWK provides high performance and scalability, while the Linux kernel provides compatibility. Multi-kernels show the promise of being able to meet tomorrow's extreme-scale computing needs while providing strong isolation, yielding high performance and scalability needed by classical HPC applications.**

McKernel and mOS started as independent research initiatives to explore the above potential. Previous work described their design and architecture advantages. This paper deploys the two LWKs and presents results from running them on a 2,048-node system with Intel® Xeon Phi™ processors (KNL) connected by Intel Omni-Path Fabric. We compare the performance of McKernel, mOS, and Linux. Although the two multi-kernel efforts approached the problem from different angles, the results show a median performance improvement of 9% with some applications as high as 280% validating the efficacy of the multi-kernel approach. We provide insight into the performance improvements and discuss the strengths of the two different multi-kernel approaches.

I. Introduction

Hardware complexity of high-end supercomputers has increased substantially over the past decade resulting in extreme degrees of multi-level parallelism, many-core CPUs, heterogeneous architectures, deepening memory hierarchies, and the growing importance of power constraints. This trend is expected to continue. At the same time, while classical high-performance computing applications are still of great interest, there has been increased interest in a diverse set of workloads including big data analytics, machine and deep learning, and multi-tenancy. This gives rise to the demand for software techniques such as in-situ analysis, work-flow composition, and the need for sophisticated monitoring and performance tools [1].

The increasing hardware complexity demands Operating Systems (OSes) that can rapidly adapt to new hardware requirements and that can support novel programming paradigms and runtime systems. Furthermore, POSIX compatibility is not enough. Increasingly, modern applications, tools, and runtime systems depend on Linux features including the `/proc` and `/sys` pseudo file systems. The combination of all the above suggest a different approach to OS structure may be valuable.

Traditional, stand-alone lightweight kernel (LWK) operating systems specialized for HPC have a proven track record of delivering the scalability and performance required by classical HPC applications, as well as providing innovative capabilities for meeting specific hardware or workload requirements [2]–[7]. However, they did not provide full support for Linux APIs and the Linux environment that have emerged as fundamental constructs that enable an increasingly diverse set of workloads.

Multi-kernels attempt to marry LWK performance and scalability with the needed Linux compatibility. It is not clear whether doing that harms LWK performance or cripples Linux compatibility. This paper is a first report to show that LWKs retain their performance benefits when run as multi-kernels.

We focus on two lightweight multi-kernels, that leverage today's many-core processors and run a lightweight kernel and a Linux kernel simultaneously. The LWK provides high performance and scalability and Linux provides the needed compatibility. While these kernels are being developed for tomorrow's computing needs, this paper focuses on evaluating them at scale with HPC applications on one of the largest systems deployed today.

IHK/McKernel at RIKEN Advanced Institute for Computational Science and mOS at Intel Corp. started as independent research initiatives to explore the multi-kernel potential. Previous work in [8] describes their design and architecture advantages. Expanding upon the work presented there, in this paper we make the following contributions:

- We describe the deployment and evaluate the results of running lightweight multi-kernels at large scale. Specifically, we ran them on 2,048 nodes of Intel® Xeon Phi™ Knights Landing (KNL) processors, up to 128k CPU cores, and compare the performance of McKernel, mOS, and Linux using a variety of applications and benchmarks.
- Although the limited availability of dedicated compute resources at large scale constrained our analysis, we describe and evaluate the impact of a number of optimizations, both targeting specific hardware features; e.g., deep

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00022

memory hierarchies, as well as application specific needs. We demonstrate that one of the strengths of lightweight multi-kernels is their ability to quickly be modified to meet new requirements.
- Despite the fact that the two multi-kernel efforts approached the problem with different designs and implementations, the results show a median performance improvement of 9% with some applications as high as 280%, validating the efficacy of the multi-kernel approach.

The rest of this paper is organized as follows. We begin by providing an overview of each multi-kernel in Section attributes II and describe some of their common attributes as well as their salient differences. We then focus on experimental evaluation in Section III and provide further discussion of that evaluation in Section IV. Section V describes related work, and Section VI draws conclusions.

II. Multi-kernels

Lightweight multi-kernels aim to provide Linux compatibility while retaining the performance and scalability of LWKs needed for HPC. This section details two approaches: IHK/McKernel from RIKEN Advanced Institute for Computational Science, and mOS from Intel Corp. After briefly exploring the multi-kernel design space we introduce each of the two projects, highlighting some of their commonalities and differences.

A. Design Space

Multi-kernels generally have three conflicting goals as exemplified in Figure 1. The main driver is to achieve the scalability and performance of LWKs (lower right corner). That can only be successful if it is combined with a high degree of Linux compatibility (top corner). This is important because many tools, libraries, and applications expect it. Users' and system administrator's productivity depend on it as well.

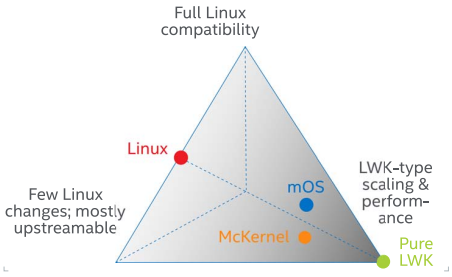


Fig. 1. Multi-kernel design space

Full Linux compatibility requires faithfully replicating system call semantics, but also mimicking the complex and ever changing pseudo file systems; e.g., `/proc`, `/sys`, that Linux uses for control and to provide information. Some of these semantics, and some POSIX mandates, are counter to what HPC applications need and make full Linux compatibility difficult to achieve.

Further complicating the picture is the need to keep track of Linux kernel developments (lower left corner). Without that, the LWK will fall behind and newer tools and features will no longer work, making the multi-kernel obsolete.

All three aspects of this struggle are symbolized as opposing corners in Figure 1. Straying too close to one corner makes it difficult to achieve the goals in the other two corners. Think of rubber bands pulling the design options into the corners. A pure, hypothetical LWK would sit at the bottom right corner. It has no Linux compatibility.

Linux, by definition, provides the necessary Linux compatibility and is the code the LWKs have to interact with. The point representing Linux is on the opposing side of a pure LWK. Rubber bands attached to the top and bottom left corners pull it there.

Multi-kernels try to find spots in between. Since they are trying to achieve all three goals represented in the triangle, the multi-kernels are pulled toward the center of the figure. Specific requirements for each project, OS design, and LWK implementation are forces that pull these kernels into unique positions in the diagram. Both mOS and McKernel emphasize LWK performance and scalability and are, therefore, closest to that corner without straying too far from the other two corners in the triangle.

McKernel is more isolated from the Linux source since it is implemented as a kernel module. Therefore, it has a stronger affinity to the left hand corner than mOS which is much more tightly integrated into the Linux kernel. Because of the latter aspect, it is easier for mOS to achieve Linux compatibility and acquire the latest features. McKernel has more of the typical LWK performance characteristics than the current version of mOS. Therefore, McKernel is closer to the LWK corner than mOS.

In addition to the three aspects discussed above, we emphasize lightweight multi-kernels' ability to easily adapt to new hardware features. This is inherent from the LWKs' small code base, which provides a fertile ground for experimentation and rapid prototyping of new ideas. As we will demonstrate it in Section III, we found this property particularly advantageous with respect to dealing with the multiple memory types available on the Xeon Phi™ processor.

B. IHK/McKernel

IHK/McKernel is a lightweight multi-kernel that comprises two main components: A low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [9] and an LWK called McKernel [10]. An architectural overview of IHK/McKernel is shown in Figure 2.

IHK provides capabilities for partitioning resources in a many-core environment; e.g., CPU cores and physical memory, and it enables management of LWKs. IHK can allocate and release host resources dynamically without rebooting the host machine. Additionally, IHK is implemented as a collection of kernel modules without any modifications to the Linux kernel code, which enables straightforward deployment on a wide range of Linux distributions. Besides resource and

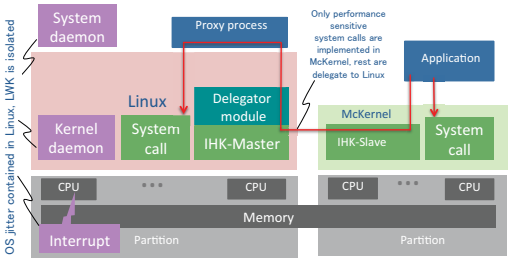


Fig. 2. Overview of the IHK/McKernel architecture and the system call offloading mechanism.

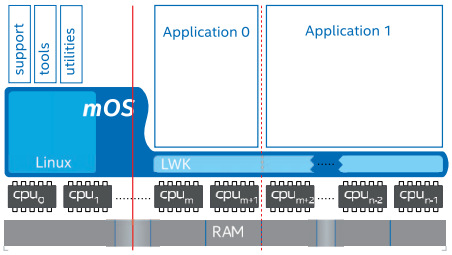


Fig. 3. Conceptual view of mOS

LWK management, IHK also provides an Inter-Kernel Communication (IKC) layer, upon which system call offloading is implemented [9].

McKernel is an LWK developed from scratch. It is primarily designed for HPC and it boots only from IHK. McKernel retains a binary compatible ABI with Linux. However, it implements only a small set of performance sensitive system calls. The rest are offloaded to Linux. Specifically, McKernel provides its own memory management, it supports multi-processing and multi-threading, it has a simple scheduler, and it implements signaling. It also enables inter-process shared memory mappings and it provides standard interfaces to hardware performance counters.

For every single process running on McKernel there is a process spawned on Linux, called the *proxy process*. The proxy process' fundamental role is to facilitate system call offloading. 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 various state information that would have to be otherwise managed by the LWK. McKernel for instance has no knowledge of file descriptors; it simply returns the descriptor 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 tracked by the Linux kernel. We emphasize that IHK/McKernel runs HPC applications on the LWK to achieve scalable execution but the full Linux API is available via system call offloading.

C. mOS

mOS follows a considerably different design and implementation path than IHK/McKernel. While the fundamental philosophy of mOS remains similar; i.e., to implement performance sensitive kernel services in the LWK code base and to rely on Linux for the rest, mOS compiles the LWK code directly into Linux.

Figure 3 shows the idea to run an LWK on the cores that run the HPC applications, while Linux runs on a few cores to the left in the drawing. As we will see in Section II-D4, mOS' structure facilitates tools support better than McKernel's.

The mOS LWK also provides its own memory management and scheduling subsystems, but the tighter Linux integration allows mOS to take advantage of Linux functionalities in a more straightforward fashion. This decision places mOS close to the LWK performance corner while allowing it to provide a high-level of Linux compatibility.

In particular, mOS' system call offloading mechanism is substantially different than that of the proxy approach. mOS retains Linux kernel compatibility at the level of its internal kernel data structures; e.g., the `task_struct`, which enables mOS to move threads directly into Linux. System call offloading is hence implemented by migrating the issuer thread into Linux, executing the system call and migrating the thread back to the LWK component.

D. Commonalities and Differences

There are many commonalities and differences between IHK/McKernel and mOS, some of which has been already explored in previous work on multi-kernel design considerations [8]. Here we further elaborate on their most important features, predominantly focusing on the ones with performance implications.

1) Node Configuration: Both IHK/McKernel and mOS provide strong partitioning of node resources; i.e., CPU cores and memory, between Linux and the LWK. In our target configuration a few CPU cores are reserved for Linux and the LWK oversees the rest of the system. Strong partitioning between the two kernels is a key property for preventing OS jitter from Linux to be propagated to the LWK. Additionally, the LWK's ability to exercise full control over its assigned resources is crucial for providing high performance. As mentioned above, HPC applications are executed on the LWK to attain performance and scalability, but full Linux compatibility is supported via system call offloading.

Both of these LWKs target high performance computing. Therefore, data locality and topology awareness are of utmost importance. Both of the LWKs are NUMA aware, not only in terms of their internal memory management policies (see below), but also in the way how they interact with Linux. mOS follows a NUMA aware mapping from LWK to Linux cores when thread migration is performed during system call offloading. Similarly, IKC, the communication framework of IHK, understands the underlying topology to perform efficient message delivery between the two kernels.

With respect to MPI applications, mOS allows LWK resources to be divided at the time of application launch. This division respects NUMA boundaries and binds threads to CPU

cores accordingly. McKernel provides a similar feature for dealing with CPU cores, although it does not partition memory between LWK processes. Generally, McKernel's philosophy is to follow a Linux compatible interface – even at the level of MPI process binding related environment variables – while mOS attempts to provide an easy to use interface.

2) Scheduling: LWKs usually provide simple scheduling policies. Both LWKs discussed in this paper employ a round-robin, non-preemptive, co-operative scheduler, as their primary purpose is to stay out of the way of applications. McKernel optionally provides time sharing, but it enables it only on specific CPU cores. It is also worth emphasizing that mOS put a significant effort into eliminating undesired kernel tasks on LWK cores which might stray there from a Linux managed core. McKernel is better isolated in that regard, since the Linux kernel cannot interact with the McKernel scheduler. As we will discuss in Section II-D4, the stricter isolation of McKernel, however, comes at the price of more significant effort for Linux compatibility.

3) Memory Management: Memory usage patterns of HPC applications are typically simpler than that of commercial applications and many features of general purpose OS kernels are not needed in HPC. For example, to completely eliminate the associated cost of page faults and to provide a more predictable behavior, both McKernel and mOS try their best to map physically contiguous memory upfront, at the time of the `mmap()` system call. An implication of contiguous physical memory is better cache performance, similar to techniques such as page coloring. Additionally, both kernels employ large pages whenever and wherever; e.g., even on the stack, it is possible, using 1 GB pages if the size of the mapping allows it. Another optimization both LWKs employ is to aggressively extend the heap to avoid contention during physical memory allocation in subsequent `brk()` calls and to ensure that large page based mappings can be established.

A strength of LWKs is their small code base that enables rapid experimentation. Considering the upcoming plethora of memory technologies and the deepening memory hierarchy, we believe that dealing with various memory types is a prime example where LWKs can potentially do better than Linux. Both mOS and McKernel provide sophisticated features to deal with the on-package high-bandwidth memory (MCDRAM) of the Xeon Phi™. Again, while mOS understands the presence of MCDRAM and provides its exclusive options to control where program sections are loaded, McKernel implements the standard NUMA APIs; e.g., the `set_mempolicy()` system call, to deal with different memories, but provides specific extensions. Ultimately, both LWKs enable fine-grain options to regulate the placement of certain process memory areas; e.g., the stack, heap or the BSS. Both kernels can also silently fall back to DDR4 RAM once they run out of MCDRAM. This is important, because standard NUMA binding policies, in combination with the way how MCDRAM is exposed in Linux in terms of NUMA distances, currently prevent an easy way to achieve this behavior in Linux in Sub-NUMA Clustering mode with four NUMA quadrants (SNC-4).

For non-file-backed mapping, both LWKs allocate physical memory at the time of the mapping request; e.g., `mmap()` or `brk()`, when physical memory to back it entirely is available. McKernel has an additional feature to automatically fall back to demand paging to allow best effort allocation from the specific NUMA domain when enough physical memory is not available. The current version of mOS is more rigid: Only physically available memory can be allocated.

4) POSIX/Linux Compatibility and Tools Support: For a multi-kernel we consider achieving full POSIX and Linux compatibility, as well as support for standard tools, such as profilers, debuggers, etc., almost as important as providing scalability and performance. The design differences between mOS and IHK/McKernel have probably the most pronounced impact on this aspect. Due to mOS' tight integration with Linux, it can provide a Linux like environment with far less effort. For example, McKernel needs to implement various `/sys` and `/proc` files to reflect the resource partition assigned to the LWK, while mOS mostly reuses the Linux implementation. Additionally, in McKernel most tools must run on an LWK core, while mOS can leave them on the Linux side. The reason is that services like `ptrace()` and `prctl()` are difficult to implement in the proxy model when crossing kernel boundaries, as opposed to mOS that can directly reuse Linux' `ptrace()` implementation.

5) Tracking Linux Changes: As mentioned above, IHK is implemented as a collection of Linux kernel modules, which makes tracking Linux kernel changes relatively easy, but has the drawback that McKernel does not become active until after Linux has booted. As a consequence, while mOS can grab large contiguous physical memory blocks early during the boot sequence, McKernel has to request them from Linux later, potentially after Linux has already placed unmovable data structures into it. On the other hand, upgrading mOS to a newer Linux kernel is a bigger effort than porting the IHK module. However, mOS ports have lasted only a couple of days each time it has been moved to a newer Linux kernel. The effort to keep both mOS and McKernel current with the latest Linux kernel is comparable to the work required to maintain a Linux device driver.

6) Application Specific Features: We mentioned that an LWK's ability to quickly adapt to new HW features is extremely beneficial. Similar arguments can be made for application specific needs. As the code base of an LWK is small[1] one can rapidly experiment with features targeting specific application needs. Complex interactions between applications, runtimes, and the OS kernel are not easy to predict or measure. They also require access to large machines for experimentation. Some of these features; e.g., the aggressive extension of the heap, can also potentially break certain POSIX requirements and may not be safe for all applications. These features are optional and can be disabled selectively at job launch time in McKernel and mOS.

[1]At the time of writing this paper, mOS modifies or adds 88 files in the Linux kernel and consists of about 20k lines of code. The same number for IHK/McKernel is approximately 70k lines of code.

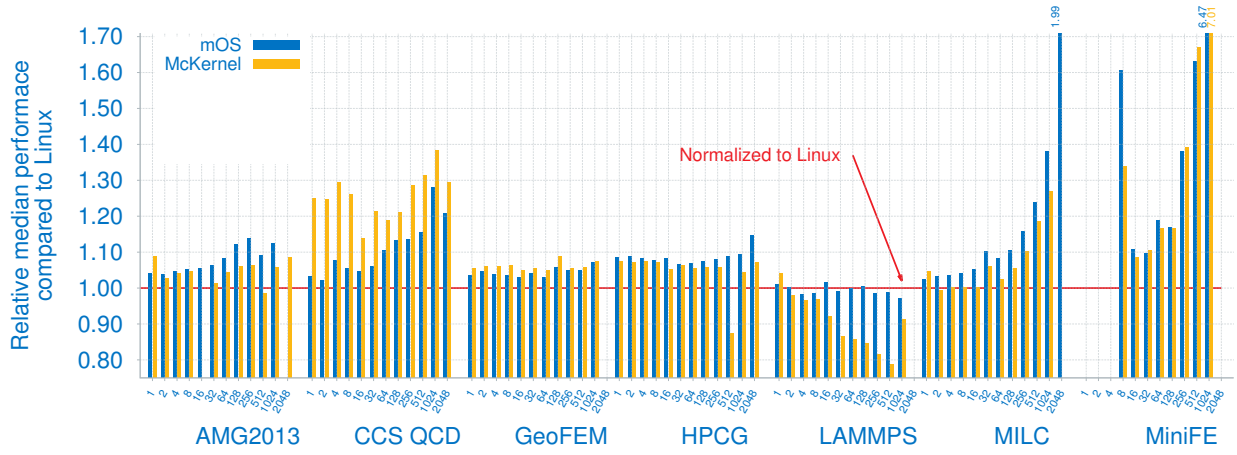


Fig. 4. Comparing mOS and McKernel against the Linux baseline

III. Evaluation

A. Experimental Environment

All of our experiments were performed on Oakforest-PACS (OFP), a Fujitsu built, 25 peta-flops supercomputer installed recently at JCAHPC organized by The University of Tsukuba and The University of Tokyo [11]. OFP is comprised of eight-thousand compute nodes that are interconnected by Intel's Omni Path network. Each node is equipped with an Intel® Xeon Phi™ 7250 Knights Landing (KNL) processor, which consists of 68 CPU cores, accommodating 4 hardware threads per core. The processor provides 16 GB of integrated, high-bandwidth MCDRAM and it also is accompanied by 96 GB of DDR4 RAM. For all experiments, we configured the KNL processor in SNC-4 flat mode; i.e., MCDRAM and DDR4 RAM are addressable at different physical memory locations and both are split into four NUMA domains. On each compute node, the operating system sees 272 logical CPUs organized around eight NUMA domains.

The software environment we used is as follows. Compute nodes run XPPSL `1.4.1` with Linux kernel version `3.10.0-327.22.2`. XPPSL is a CentOS based distribution with a number of Intel provided kernel level enhancements specifically targeting the KNL processor. We used Intel MPI 2018 Beta Build 20170209 that offers a few unreleased improvements for parallel job spawning.

For all experiments, we dedicated 64 CPU cores to the application and reserved 4 CPU cores for OS activities. This is a common scenario for OFP users where daemons and other system services run on the first four cores. Many applications need a power of two number of nodes, or do not run faster on 66 or 68 cores. Additional experiments have shown that mOS using 64 or 66 cores beats Linux on 68 cores. This is often due to CPU 0 running services and introducing noise.

We emphasize that for the Linux runs we used the Fujitsu's HPC optimized production environment, e.g., application cores were configured with the `nohz_full` Linux kernel argument to minimize operating system jitter. For the McKernel measurements we deployed IHK and McKernel, commit hash `7b3872ed` and `c32b1ada`, respectively. We utilized IHK's resource partitioning feature to reserve CPU cores and physical memory dynamically. As for mOS, we deployed a port of the mOS LWK (version `3.10.0-hfi141-tmusta-mos-rc2-gb57bbbd`) which is based on the Linux kernel mentioned above.

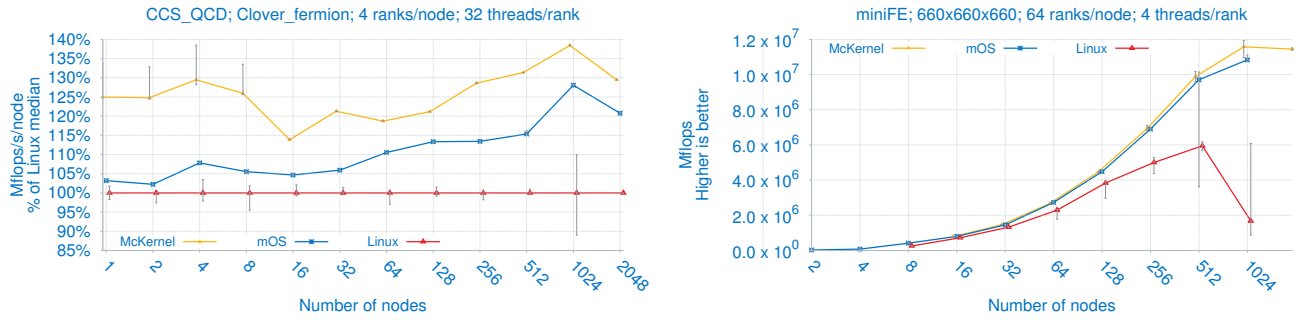
B. Workloads and Methodology

We chose a number of workloads from the CORAL benchmark suite as well as applications used for evaluation during the procurement of the OFP machine. These workloads are based on availability, in-house expertise to configure and run them, user demands, and suitability for the type of system these OS projects are targeting. We went to great lengths to provide good settings for Linux to ensure a fair comparison.

Neither mOS nor McKernel have ever run before at this scale. We expected problems when jumping that many orders of magnitude in node and core count, and experienced our fair share. Nevertheless, we were able to successfully collect data for eight application benchmarks: AMG 2013 [12], CCS-QCD [13], GeoFEM [14], HPCG [15], LAMMPS [16], Lulesh 2.0 [17], MILC [18], and MiniFE [19]. For detailed information on the benchmarks, their configuration and the exact runtime arguments we used visit the following URL:
http://www.sys.aics.riken.jp/Members/bgerofi/ipdps18/addendum.html

All applications, except MiniFE ran weakly scaled. All but CCS-QCD were sized to fit entirely into MCDRAM. We chose different configurations and usage scenarios to help us understand the behavior of these OSes and test them under varying conditions.

As for CCS-QCD, the two LWKs were able to load a portion of the workload into MCDRAM and then seamlessly spill the rest into DDR4 RAM. Linux is only capable of doing this in quadrant mode, but not in SNC-4 mode. Under Linux we ran CCS-QCD out of DDR4 RAM only.



(a) CCS-QCD scaling as a percentage compared to Linux

(b) MiniFE scaling experiments

Fig. 5. CCS-QCD and MiniFE scaling results

Although many KNL clusters are configured to run in quadrant mode because it allows exploitation of the higher bandwidth of MCDRAM with less tuning effort, SNC-4 mode offers the highest possible hardware performance.

C. Scalability and Performance

Figure 4 summarizes most of the results we gathered in a single plot comparing mOS and McKernel results against the Linux baseline. We left out Lulesh 2.0 since it uses different node counts than the rest of the benchmarks. We show detailed Lulesh 2.0 results in Figure 6a.

For each application in Figure 4 we show the result as reported by the benchmark for 1 through 2,048 nodes. We ran most applications five times and show the median. For clarity, we clip the y-axis below 0.75 and above 1.75. Measurements for MILC, and MiniFE that are beyond that range, are marked with the measured median values. The scaling graphs also show error bars indicating the maximum and minimum values we measured.

Since we used 64 physical cores per node, the node counts in Figure 4 represent physical CPU core counts from 64 to 131,072. Most applications ran with 32 or 64 MPI ranks per node. Due to limited space, we cannot present all application results in detail. We highlight CCS-QCD, MiniFE, LAMMPS, and Lulesh 2.0 as interesting examples representing all of our applications.

For CCS-QCD we chose a large problem size that does not fit into MCDRAM. Both McKernel and mOS allow allocation of physical MCDRAM pages until that resource runs out. After that, further allocations are satisfied with DDR4 RAM memory. The switch-over is transparent and seamless. Achieving this in Linux is more difficult. In quadrant mode, the `numactl -p` option can be used by specifying MCDRAM as the preferred NUMA domain. In SNC-4 mode, four such domains exist, but the current Linux implementation allows only one to be listed. We chose to use DDR4 RAM only for CCS-QCD when running on Linux to highlight the LWK's capabilities for dealing with complex memory hierarchies. The excellent results in Figure 5a, up to 39% and 28% improvement on McKernel and mOS, respectively, are partially attributable to the automatic use of MCDRAM by the LWK even in situations when not all data fits into MCDRAM. While we did not include it in the plot, we did experiment with running CCS-QCD exclusively out of DRAM on McKernel as well, for which we observed approximately 5% slowdown when running on 2,048 nodes. For further information on why the McKernel results are better than mOS see Section IV.

MiniFE stands out as the application that ran almost seven times faster on the LWK than on Linux on 1,024 nodes. As Figure 5b illustrates, that apparent performance gain is actually due to Linux performance dropping precariously. Further large-scale experiments will be needed to determine the exact reasons for this, although we know that MiniFE is sensitive to the performance of MPI collective operations; e.g., `MPI_Allreduce()`, which typically benefit from jitter-less operating system kernels.

A similar drop-off at a high node count occurred with Lulesh 2.0. Note that this is not a single outlier. The 1,728-node Linux result in Figure 6a is the median of five experiments. The Lulesh 2.0 experiments are also interesting due to the unexpectedly large difference between the Linux and mOS results. We detail in Section IV some of the reasons behind this mystery.

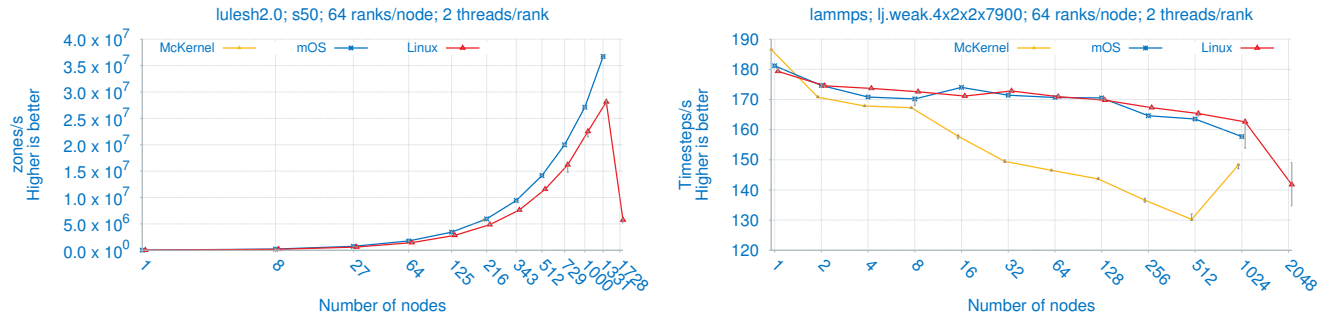
Although the LWKs usually performed better than Linux, sometimes significantly so, there are also examples where the gains were minimal or Linux was actually better. LAMMPS is one example as shown in Figure 6b. Again, Section IV will provide more information for this application.

D. Linux Compatibility

One strong justification for the use of multi-kernels is POSIX and Linux compatibility. It is therefore important to assess how these two projects fare in that regard.

Unfortunately, measuring compatibility is not simple. At first glance, the Linux Test Project (LTP) suite of tests [20] would seem a good starting point. Concentrating only on system calls, McKernel passes all but 32 of them. For mOS the numbers are more bleak: 111 tests out of 3,328 fail.

McKernel has been using LTP as a regression test since the early days of its development. Of the 32 tests that still fail,



(a) Lulesh 2.0 scaling experiments

(b) LAMMPS scaling experiments

Fig. 6. Lulesh 2.0 and LAMMPS scaling results

some are not supported intentionally, mainly because of the nature of HPC workloads, others are simply missing implementation. Eleven of the 32 failing experiments attempt to test various combinations of the `move_pages()` system call, which is work in progress. Another representative experiment tests the error behavior of an unusual `clone()` flag combination, which actual applications never seem to use.

Many of the LTP tests rely on `fork()` to set up the experiment. In mOS, `fork()` is not fully implemented yet which results in many failures before the tests of the targeted system calls even begin. As with McKernel, there are experiments that test esoteric behavior and fail in mOS. For example, `ptrace()` is working in mOS. However, four of the five `ptrace()` experiments fail.

In Section IV we describe an HPC-optimized implementation of `brk()`. Because mOS does not return memory to the system when the heap shrinks, tests that expect a page fault fail. Such a test looks for Linux behavior that HPC applications do not need or expect.

IV. Discussion

This section provides further information on some of the application results presented in Section III. Specifically, we discuss Lulesh 2.0, CCS-QCD and LAMMPS in detail.

The significant performance improvement of Lulesh 2.0 shown in Figure 6a comes from the overhead of the `brk()` system call. Both LWKs have HPC-optimized implementations for `brk()`. In mOS this feature can be toggled by a runtime option, in McKernel it is currently implemented in a separate branch, but IHK allows booting different kernel images per application. Table I shows figures for the case where memory is taken only from DDR4 RAM. Note that the figures do not directly correspond to the results we obtained when we evaluated Lulesh 2.0 with the `brk()` optimizations enabled, because it runs out of MCDRAM. This is because we encountered a bug in the feature toggling code when performing the comparison.

Nevertheless, Table I shows that about 6% of the performance gain we observed in Lulesh 2.0 stems from mOS features unrelated to heap management, while an additional

TABLE I
Lulesh performance in DDR4 RAM with and without `brk()` optimizations

Linux	8,959 zones/s	100.0%
mOS, heap management disabled	9,551 zones/s	106.6%
mOS, regular heap management	10,841 zones/s	121.0%

15% can be attributed to the handling of `brk()` in mOS. We ran some additional experiments with a smaller workload, `-s 30` instead of `-s 50` in Figure 6a, to analyze the behavior of `brk()`. There were 7,526 queries – calling `sbrk()` with a value of 0 – 3,028 expansion requests, and 1,499 requests for contraction for a total of about 12,000 calls to `brk()` in the few seconds the program runs with the small workload. At its largest, the heap grew to 87 MB, but looking only at the growth requests, the cumulative amount of memory requested was 22 GB. Under Linux this results in a lot of page faults, and it is happening on 64 MPI ranks on each node.

The `brk()` optimizations in mOS are as follows. The heap is aligned to a 2 MB address boundary. When it grows, it expands in 2 MB increments. Shrink requests are ignored because we found that many high-end HPC applications allocate memory at the beginning and retain it until the end of the run. Physical memory pages are allocated at the time `brk()` is called. That means subsequent accesses to the heap do not cause page faults in the LWKs. Finally, upon a growth request and allocation of a new 2 MB page, only the first 4 kB are zeroed. mOS clears the first 4 kB because of a bug in AMG 2013, or one of the libraries it uses, that erroneously expects heap memory to be cleared upon return from `brk()`.

Clearing memory for `malloc()` is not a POSIX requirement; `calloc()` should be used for that. Nevertheless, Linux initially maps the zero page and performs a page fault and clear on first write access to that page. Further hampering Linux is that it can only allocate large pages when the heap boundary happens to be properly aligned and the request is large enough.

These LWK optimizations – McKernel employs similar strategies – explain the large performance improvement for Lulesh 2.0. The LWKs avoid page faults and make automatic

and consistent use of large pages. These types of optimizations also show why it is difficult for Linux to become an optimized OS for high-end HPC. Growing the heap 2 MB at a time, not relinquishing memory when it shrinks, and allocating physical pages at the time of mapping are not suitable strategies for applications that make many small requests and have to compete for node resources with a lot of other unrelated tasks.

We are turning our attention to the CCS-QCD workload shown in Figure 5a. Surprisingly, McKernel performed visibly better than Linux and mOS. Although we could not verify the reason, we found clues that may shed light on what was happening. This application was configured to use a large amount memory that didn't fit into MCDRAM entirely, and as mentioned in Section II, McKernel can fall back to demand paging when there is not enough physically contiguous memory to cover a virtual memory range.

Based on kernel logs taken during the experiments we found that some of the ranks, but not all, reported falling back to demand paging on certain memory ranges. Our hypothesis is that with demand paging enabled, ranks inside the node could better utilize MCDRAM as opposed to dividing memory resources upfront, which is what mOS does by default. Unfortunately, due to excessive demand for the machine we ran our measurements on, we were unable to do further investigation before writing this paper.

The last application we discuss in more detail is LAMMPS. As shown in Figure 6b, neither mOS nor McKernel performed better than Linux at scale, despite the fact that single node results were promising. To find out why, we profiled both the application and the kernel. It turns out that the Intel Omni-Path network involves system calls for certain operations and LAMMPS utilizes communication routines that rely on those. This introduces extra latency and drop in network bandwidth when running on McKernel, because system calls on device files are offloaded to Linux. However, we consider this as temporary shortcoming of OmniPath's current generation as most high-performance networks are usually driven entirely from user-space. We are still investigating the reasons for mOS, but we note that both McKernel and mOS are still in the early stages of their life-cycles and in active development.

Finally, we provide insight into two additional optimizations McKernel offers via command line options to to the proxy process. The first option, `--mpol-shm-premap`, enables pre-mapping shared memory sections used by the MPI implementation for intra-node communication. This helps avoiding contention in the page fault handler. The second optimization we applied for a number of benchmarks is `--disable-sched-yield`, which injects a shared library into the application to hijack glibc's `sched_yield()` system call and simply ignores it. This helps to eliminate user/kernel mode switches, if the call is in the performance sensitive execution path of an algorithm. Although we had no chance to obtain performance breakdown regarding the impact of these calls at scale, with the combination of these two we observed 9% and 2% improvements on 16 nodes for AMG 2013 and MiniFE, respectively.

V. Related work

McKernel and mOS are considered multi-kernels. Before we compare them to other multi-kernel projects, we have a look at lightweight kernels which preceded them.

A. Lightweight Kernels

Lightweight kernels [21] designed for HPC workloads date back to the early 1990s. The authors of [22] provide reasons for their use, design principles, and a history of early LWKs developed at Sandia National laboratories. One example in the series is Catamount [5], which has been successfully deployed on a large scale supercomputer. IBM's BlueGene line of supercomputers have also been running an HPC specific LWK called the Compute Node Kernel (CNK) [6]. While Catamount has been developed from scratch, CNK borrows a significant amount of code from Linux; e.g., glibc and NPTL, so that it can better comply with standard Unix features. The most recent of Sandia National Laboratories' LWKs is Kitten [7], which distinguishes itself from their prior LWKs by providing a more complete Linux-compatible environment. It also provides virtual machine monitor capability via Palacios [23] that allows unmodified guest OSes to run on top of Kitten. Despite all these efforts, with the ever growing demand for full Linux/POSIX feature compatibility from the application side, it has become increasingly difficult to provide support for all Linux features without compromising the fundamental goal of retaining scalability and performance.

Instead of building an LWK from the ground up, another approach is to start from Linux and modify it to meet HPC requirements. The modifications are done to ensure low noise, scalability and predictable application performance. Cray's Extreme Scale Linux [2], [3] and ZeptoOS [4] follow this path. The usual approach is to eliminate daemon processes, simplify the scheduler, and replace the memory management system. There are two main issues with the Linux based approach. First, the excessive modifications occasionally break Linux compatibility, which is undesirable. Second, because HPC usually follows, or sometimes dictates, rapid hardware changes that need to be reflected in the kernel code, the modified Linux kernels tend to diverge from the main Linux tree which results in an endless need to maintain kernel patches. In contrast, both mOS and IHK/McKernel strive to attain full Linux compatibility without sacrificing LWK performance.

B. Operating Systems for Multi-core CPUs

The primary design concern of the K42 [24] research project was scalability. Similar to how mOS and IHK/McKernel selectively implement a set of performance sensitive system calls, K42 enabled applications to bypass the Linux APIs and call the native K42 interfaces directly. However, it involved a significant entanglement with Linux which made it cumbersome to keep up with the latest Linux modifications. While mOS and McKernel also rely on Linux, one of their primary design criteria was to minimize the effort required to keep up-to-date with the rapidly moving Linux kernel.

Multi-kernels in commercial computing have also been proposed. Tessellation [25] and Multikernel [26] are driven by the observation that modern computers already resemble a networked system and so the OS should also be modeled as a distributed system. The Tessellation project [25] proposed Space-Time Partitions, an approach that partitions resources by dividing CPU cores into groups called cells. Each cell is responsible for some system services or for a particular application. Because applications and system services can be assigned to distinct cells, Tessellation's structure is similar to both mOS and IHK/McKernel, where HPC workloads are bound to LWK cores while system daemons reside on CPU cores managed by Linux.

Multikernel [26] runs a small kernel on each CPU core and OS services are built as a set of cooperating processes. Each process is running on one of the multi-kernels and communicates using message passing. Similarly, IHK/McKernel relies on a message passing facility that allows communication between the two types of kernels, and consequently between the application and its Linux proxy process.

Zellweger et. al have recently proposed decoupling CPU cores, kernels and operating systems [27]. Their system enables applications to be migrated to a separate OS node while the kernel is updated on the original CPU core. In mOS, process representation in the LWK retains compatibility with Linux kernel data structures so that it can directly migrate threads for system call offloading. This mechanism is similar to the idea of decoupling application state from the operating system, although in mOS the primary purpose is to execute certain system calls in a different kernel context where a richer set of Linux features is available.

C. Multi-kernels in HPC

The idea of multi-kernels in the HPC context has also been studied for a number of years. FusedOS [28] was the first system to combine Linux with an LWK. FusedOS' primary objective was to address core heterogeneity between system and application cores and at the same time to provide a standard operating environment. Contrary to mOS and McKernel, FusedOS runs the LWK at user level. The kernel code on application CPU cores 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 this CL process that runs on Linux. Consequently, FusedOS provides the same functionality with the Blue Gene CNK from which CL was derived. The FusedOS work was the first to demonstrate that Linux noise can be isolated to the Linux cores to avoid interference with the HPC application running on the LWK CPUs. This property has been one of the main drivers for both mOS and McKernel.

Hobbes [29] was one of the projects in DOE's Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to support application composition, which is emerging as a key approach to address scalability and power concerns anticipated in future extreme-scale architectures. Hobbes utilizes virtualization technologies to provide the flexibility to support requirements of application components for different node-level operating systems and runtimes. The Kitten [7] LWK forms the base layer of Hobbes and Palacios [23], running on top of Kitten, serves as a virtual machine monitor.

Argo [30] is another DOE OS/R project targeted at applications with complex work-flows. While Argo originally also targeted a multi-kernel based software architecture, it recently turned toward primarily relying on container technologies. Currently, it investigates how to enhance the Linux kernel's container framework so that it can meet HPC requirements.

Recent efforts have also demonstrated multi-kernel's ability of performance isolation [31], [32], an increasingly important aspect of system software as we move toward multi-tenant deployments. However, the above studies were performed on considerably small scale.

VI. Conclusion and future work

Lightweight multi-kernel operating systems in HPC leverage today's many-core processors to run multiple OS kernels, typically a lightweight kernel and a Linux kernel, simultaneously. The LWK provides high performance and scalability while Linux provides the required compatibility for supporting tools and the full POSIX/Linux APIs.

To the best of our knowledge, this is the first study that has successfully demonstrated the viability of the multi-kernel approach at scale. By deploying and evaluating two multi-kernels on up to two thousand Intel® Xeon Phi™ Knights Landing (KNL) nodes, we observed a median performance improvement of 9% with some applications as high as 280%. We have provided insight into the performance gains and compared the strengths of the two different multi-kernels approaches. One of our key findings is that besides the LWKs ability to provide scalability and performance, the multi-kernel approach enabled us to rapidly experiment with specific features targeting new hardware and application needs. With the increasing complexity of high-end hardware and the growing diversity of workloads, we believe this property will be highly beneficial in the future.

mOS and IHK/McKernel are also in an unusual position from a deployment point of view. Since both are targeted for production environments, there is plenty of future work ahead. The developers of both IHK/McKernel and mOS are working on improving kernel stability and are further exploring features that will benefit applications on extreme-scale systems.

Acknowledgments

IHK/McKernel is partially funded by MEXT's program for the Development and Improvement for the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities.

We thank The University of Tokyo and The University of Tsukuba for letting us access the Oakforest PACS machine and for their help with getting the experiments done.

We acknowledge Tomoki Shirasawa, Gou Nakamura and Ken Sato from Hitachi for their McKernel development efforts.

Tom Musta and David van Dresser visited Japan to run the first set of experiments. Tom and Andrew Tauferner conducted post-mortem experiments that are included in this paper. Thomas Spelce was invaluable in preparing the applications and our experiments, while Mike Blocksome was a great help with MPI related issues. Other members of the mOS team not listed so far contributed time and assisted in the runs and made mOS possible in the first place: John Attinella, Sharath Bhat, Jai Dayal, and Lance Shuler.

REFERENCES

[1] BDEC Committee, "The BDEC "Pathways to convergence" report," http://www.exascale.org/bdec/, Mar. 2017.

[2] 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.

[3] 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 Cray User Group*, ser. CUG, 2012.

[4] 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 the 2009 Intl. Conference on Parallel Processing Workshops*, ser. ICPPW. IEEE Computer Society, 2009, pp. 65–72.

[5] S. M. Kelly and R. Brightwell, "Software architecture of the light weight kernel, Catamount," in *Cray User Group*, 2005, pp. 16–19.

[6] 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 the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2010.

[7] K. T. Pedretti, M. Levenhagen, K. Ferreira, R. Brightwell, S. Kelly, P. Bridges, and T. Hudson, "LDRD final report: A lightweight operating system for multi-core capability class supercomputers," Sandia National Laboratories, Technical report SAND2010-6232, Sep. 2010.

[8] 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.

[9] 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 *21th Intl. Conference on High Performance Computing*, ser. HiPC, Dec. 2014.

[10] 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 *13th Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2013.

[11] Joint Center for Advanced HPC (JCAHPC), "Basic specification of Oakforest-PACS," http://jcahpc.jp/files/OFP-basic.pdf, Mar. 2017.

[12] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Appl. Num. Math.*, vol. 41, pp. 155–177, 2002.

[13] K.-I. Ishikawa, Y. Kuramashi, A. Ukawa, and T. Boku, "CCS QCD application," https://github.com/fiber-miniapp/ccs-qcd, Mar. 2017.

[14] K. Nakajima, "Parallel iterative solvers of GeoFEM with selective blocking preconditioning for nonlinear contact problems on the Earth Simulator," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC. New York, NY, USA: ACM, 2003.

[15] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark: A new metric for ranking high performance computing systems," University of Tennessee, Electrical Engineering and Computer Science Department, Tech. Rep. UT-EECS-15-736, Nov. 2015.

[16] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," San Diego, CA, USA, pp. 1–19, Mar. 1995.

[17] I. Karlin, J. Keasler, and R. Neely, "LULESH 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[18] NERSC, "MILC," http://www.nersc.gov/research-and-development/apex/apex-benchmarks/milc/, Oct. 2017.

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[20] S. Modak, B. Singh, and M. Yamato, "Putting LTP to test - validating the Linux kernel and test cases," in *Proceedings of the 2009 Montreal Linux Symposium*, Montreal, Canada, Jul. 2009.

[21] R. Riesen, A. B. Maccabe, B. Gerofi, D. N. Lombard, J. J. Lange, K. Pedretti, K. Ferreira, M. Lang, P. Keppel, R. W. Wisniewski, R. Brightwell, T. Inglett, Y. Park, and Y. Ishikawa, "What is a lightweight kernel?" in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS. New York, NY, USA: ACM, 2015.

[22] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 6, pp. 793–817, Apr. 2009.

[23] 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 *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010.

[24] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: Building a complete operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 133–145, Apr. 2006.

[25] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client OS," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, ser. HotPar, 2009.

[26] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP, 2009, pp. 29–44.

[27] G. Zellweger, S. Gerber, K. Kourtis, and T. Roscoe, "Decoupling cores, kernels, and operating systems," in *11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, Broomfield, CO, Oct. 2014, pp. 17–31.

[28] 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 *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, Oct. 2012, pp. 211–218.

[29] 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 the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS, 2013.

[30] P. Beckman, M. Snir, P. Balaji, F. Cappello, R. Gupta, K. Iskra, S. Perarnau, R. Thakur, and K. Yoshii, "Argo: An exascale operating system," http://www.mcs.anl.gov/project/argo-exascale-operating-system, Mar. 2017.

[31] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti, "Achieving performance isolation with lightweight co-kernels," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 149–160.

[32] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 1041–1050.