Linux vs. Lightweight Multi-kernels for High Performance Computing: Experiences at Pre-Exascale

Balazs Gerofi[†], Kohei Tarumizu[‡], Lei Zhang[‡], Takayuki Okamoto[‡], Masamichi Takagi[†], Shinji Sumimoto[‡], Yutaka Ishikawa[†]
[†]RIKEN Center for Computational Science, JAPAN
[‡]Fujitsu Ltd., JAPAN
{bgerofi, masamichi.takagi, yutaka.ishikawa}@riken.jp, {tarumizu.kohei, zhang.lei, tokamoto, sumimoto.shinji}@fujitsu.com

ABSTRACT

The long standing consensus in the High-Performance Computing (HPC) Operating Systems (OS) community is that lightweight kernel (LWK) based OSes have the potential to outperform Linux at extreme scale. To explore if LWKs live up to their expectation we developed *IHK/McKernel*, a lightweight multi-kernel OS designed for HPC, and deployed it on two high-end supercomputers to compare its performance against Linux. Oakforest-PACS, an Intel Xeon Phi (x86) based supercomputer, runs a moderately tuned Linux distribution. Fugaku, the world's fastest supercomputer at the time of writing this paper, is based on Fujitsu's A64FX (aarch64) CPU that runs a highly tuned Linux environment.

We discuss recent developments in our OS and provide a detailed description on the challenges of tuning Fugaku's Linux for high-end HPC. While in a moderately tuned environment McKernel significantly outperforms Linux (by up to approximately 2X), on Fugaku we observe an average of 4% speedup across all our experiments, with a few exceptions where the LWK outperforms Linux by up to 29%. As part of our evaluation we also disclose a full scale (158,976 compute nodes) noise profile of the Fugaku system.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Operating Systems** → Organization and Design.

KEYWORDS

high-performance computing, operating systems, lightweight kernels, multi kernels, scalability

ACM Reference Format:
Balazs Gerofi[†], Kohei Tarumizu[‡], Lei Zhang[‡], Takayuki Okamoto[‡], Masamichi Takagi[†],, Shinji Sumimoto[‡], Yutaka Ishikawa[†]. 2021. Linux vs. Lightweight Multi-kernels for High Performance Computing: Experiences at Pre-Exascale. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3458817.3476162

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476162

1 INTRODUCTION

Lightweight kernel (LWK) [43] based operating systems (OS) designed for high-performance computing (HPC) workloads have been successfully deployed in the past on a number of large-scale supercomputers. For example, Cougar [8] and Catamount [29], two LWKs that originate from the SUNMOS [45] and PUMA [49] OSes developed at Sandia National Laboratories and the University of New Mexico, were the default operating systems on the compute partitions of the ASCI Red and Red Storm supercomputers, respectively. IBM's Compute Node Kernel (CNK) [20, 33], another notable lightweight kernel, was the default OS on the BlueGene line of supercomputers.

Lightweight kernels provide excellent scalability, predictable performance, *i.e.*, very low OS jitter [40, 43], and present opportunities for rapid experimentation with novel OS concepts due to their relatively simple code base [7, 8, 14, 37, 42], something the OS community has for a long time recognized as *nimbleness* [13].

However, the lack of device driver support and the limited compatibility with the standard POSIX/Linux APIs in LWKs have prohibited their wide-spread deployment. For example, it has been reported that the main obstacle for carrying out large-scale evaluation of Kitten [31], the latest of the Sandia line of lightweight kernels, has been the lack of support for Infiniband networks [37]. At the same time, neither Catamount nor the IBM CNK provided full compatibility for a POSIX compliant `glibc`, limiting the availability of standard system calls, such as `fork()` [20]. The lack of full POSIX/Linux support has been increasingly becoming problematic with the shift to more diverse workloads in supercomputing environments, *e.g.*, Big Data analytics and machine learning [4].

Multi-kernel operating systems have been proposed to address the aforementioned shortcomings of LWKs [17, 19, 30, 37, 38, 50], where Linux and a lightweight kernel run side-by-side on compute nodes with the motivation to provide LWK scalability, to retain full compatibility with the Linux/POSIX APIs, and to reuse device drivers from Linux at the same time. Another advantage of multi-kernels is performance isolation, a property that has become increasingly desired with the emergence of many-core CPUs and co-location of different workloads [18, 37].

Linux, on the other hand, has also improved throughout the years. In particular, the introduction of Linux control groups (*i.e.*, `cgroups`), which forms one of the pillars of application containers, contributed significantly to Linux' ability for providing predictable performance and better workload isolation [52]. While previous work has shown that lightweight multi-kernels can outperform Linux on various HPC applications [14], as of today, there has been

no evaluation that compared multi-kernels with a highly tuned Linux environment.

We developed IHK/McKernel, a lightweight multi-kernel OS designed for HPC, which we deployed on two high-end supercomputers to compare its performance against Linux. While on Oakforest-PACS, an Intel Xeon Phi (x86) based supercomputer, only a moderately tuned Linux environment is available, we went to great lengths to optimize Linux for scalability on Fugaku. Fugaku is the world's fastest supercomputer at the time of writing this paper, which is based on Fujitsu's A64FX (aarch64) processor. This paper describes the specific countermeasures we took to eliminate OS jitter on Fugaku's Linux environment and reports on our findings how the lightweight multi-kernel fares against the aforementioned Linux environments.

Specifically, we make the following contributions:

- We describe the measures we took for scaling Linux to Fugaku's over 150k compute nodes. In particular, we discuss containers, virtual NUMA nodes, various techniques for OS noise mitigation including ARM64's remote Translation Lookaside Buffer (TLB) invalidation issue, and the integration of `cgroups` with hugeTLBfs.
- We present the latest developments in IHK/McKernel, the lightweight multi-kernel OS we developed and deployed on two high-end supercomputers.
- We evaluate IHK/McKernel against the two Linux environments using micro-benchmark as well as a number of different HPC applications.
- As part of the evaluation, we provide a full-scale noise profile of the Fugaku Linux environment.

To the best of our knowledge, this is the first time a lightweight multi-kernel has been compared against a highly tuned Linux environment for large-scale scientific computing. We find that while IHK/McKernel consistently outperforms the moderately tuned Linux environment on Oakforest-PACS, on Fugaku we observe an average of 4% performance improvement across all our measurements, with a few exceptions where the LWK outperforms Linux by 29%.

The rest of this paper is organized as follows. We begin with motivating our study in Section 2, which is followed by an overview of our target platforms in Section 3. The challenges of scaling Linux on the Fugaku system is described in Section 4. Section 5 provides an overview of IHK/McKernel OS and discusses recent developments. Evaluation is provided in Section 6. Section 7 discusses related work, and finally, Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

The most prevalent runtime behavior of parallel scientific applications executed on large-scale supercomputers is alternating between phases of computation and communication, also known as bulk-synchronous parallel execution [48]. While there have been various efforts to create runtime systems (*e.g.*, task-based, adaptive runtimes [3]) that avoid such behavior, as of today, bulk-synchronous execution still dominates most HPC applications. For example, eight out of the nine applications representing the nine priority issues identified during the Fugaku development project follow the bulk-synchronous parallel execution model [46].

When running bulk-synchronous applications on large-scale supercomputers, operating system management tasks may delay the execution of application processes, which adversely affects overall performance. Delay in application execution due to non-application processes, such as OS daemons, kernel daemons, interrupt processing, etc., are collectively called operating system noise, also known as OS jitter [40]. Figure 1 shows the effect of OS noise on parallel applications.

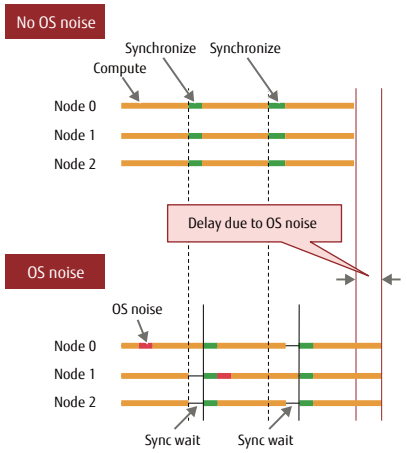


Figure 1: Impact of OS noise on bulk-synchronous parallel applications

The delay due to OS noise for bulk-synchronous applications can be estimated as the maximum length of the noises happening in the aggregated synchronization interval, which is calculated as the synchronization interval of the application times the number of threads [9]. Let us group the noises by their lengths and denote by L_i the length of the i-th group, by M the number of groups, by N the number of threads, by I_i the noise occurrence interval of the i-th group and by S the synchronization interval of the application. The delay can be estimated as follows:

$$\max_{i=1}^{M} \left(\left(1 - \left(1 - \frac{S}{I_i} \right)^N \right) \times \frac{L_i}{S} \right) \tag{1}$$

For example, OS noise could slow down an application with $N = 100,000$ threads with $S = 250$ µs synchronization interval by 20% with a machine with only one noise group with $L_1 = 1$ ms and $I_1 = 500$ s. Therefore, reducing OS noise is essential for large-scale supercomputing environments.

The operating system community in high-performance computing has spent decades on addressing OS noise mitigation and various approaches have been proposed throughout the years [13]. Due to its widely available, standard development environment and excellent tools support Linux has emerged as the dominant OS for large-scale supercomputers. Although the expectation that lightweight kernel based OSes have the potential to outperform Linux at extreme scales endured, how such an approach performs in comparison to a highly tuned Linux environment remains an open question.

Table 1: Overview of platforms and Linux runtime settings

Platform / Attribute		Oakforest-PACS [24, 26]	Fugaku [12, 44]
Node level	CPU model	Intel Xeon Phi 7250 Knights Landing (KNL)	Fujitsu A64FX
	Instruction set architecture	`x86_64`	ARM `aarch64`
	Number of CPU cores	68, 4-way SMT	50 (or 52, depending on the node), no SMT
	Number of TLB entries	L1: 64, L2: 64	L1: 16, L2: 1,024
	Memory	96GB DDR4 & 16GB MCDRAM	32 GB HBM2
	Linux distribution	CentOS 7.3	RedHat Enterprise Linux 8.3
	Linux kernel version	`3.10.0-693.11.6`	`4.18.0-240.8.1.el8_3`
	Containerization	No	Docker
	`nohz_full` on app cores	Yes	Yes
	CPU isolation	No	`cgroups`
	IRQ steering	No	Routed to OS cores
	Large page support	THP	HugeTLBfs
System level	Peak performance	25 PFlops	488 PFlops
	Number of compute nodes	8,192	158,976
	Interconnection network	Intel OmniPath	Fujitsu TofuD

3 PLATFORMS

In order to lay the groundwork for discussion on operating system issues to extreme scale HPC, we first provide an overview of the platforms used in this study with a special emphasis on their system software environment. Table 1 summarizes various hardware and software attributes of the two platforms.

3.1 Oakforest-PACS

Oakforest-PACS (OFP) is a 25 peta-flops supercomputer installed at JCAHPC, managed by The University of Tsukuba and The University of Tokyo [26]. 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. From the operating system's perspective there are 272 logical CPUs.

The software environment on OFP is as follows. Compute nodes run CentOS `7.3` with Linux kernel version `3.10.0-693.11.6` that support various Intel provided kernel level improvements specifically targeting the KNL processor. The OFP operating system environment does not strictly divide processor cores into system versus application dedicated partitions. While there is a designated group of 256 logical CPU cores that users are encouraged to use for applications, the entire chip is available for applications if desired. Application cores, nevertheless, are configured with the `nohz_full` Linux kernel argument to disable kernel timer interrupts and thus minimize operating system jitter. We note that device IRQs are balanced across the entire chip and are not restricted to cores on which timer interrupts are enabled. Previous work has shown that pinning application threads to the processor cores with no timer interrupts yields significant performance benefits [14].

The primary runtime environment for HPC applications on OFP is Intel MPI with the GNU standard library and the Linux transparent hugepages facility is enabled to reduce TLB misses.

3.2 Fugaku

Fugaku is a Fujitsu built supercomputer installed at RIKEN Center for Computational Science [44]. At the time of writing this paper, Fugaku is number one on the TOP500 list of the world's fastest supercomputers. It offers a theoretical peak double precision performance of 488 peta-flops. Fugaku is comprised of 158,976 compute nodes that are interconnected by the Fujitsu TofuD network. Each node is equipped with a Fujitsu A64FX processor, which consists of up to 52 CPU cores and provides the ARM64 (aarch64) instruction set architecture (ISA). The processor is integrated with 32GB HBM2 high-bandwidth memory.

Compute nodes on Fugaku run Red Hat Enterprise Linux (RHEL). 48 CPU cores can be used by applications and the remaining CPU cores (also referred to as *assistant cores*) are dedicated to system activities. Note that most compute nodes on Fugaku are equipped with only 50 CPU cores, on those nodes the number of OS cores in limited to two. Application cores are not only configured with the `nohz_full` Linux kernel argument to minimize operating system jitter, but are also separated from system processes (*e.g.*, daemon processes) using the Linux control groups (cgroups) facility. For further details on Fugaku's software stack and the approach we took to scale Linux for HPC see Section 4.

One notable difference between the Xeon Phi and A64FX is the number of TLB entries supported. As shown in the table, while Xeon Phi has only 64 last level TLB entries, A64FX provides 1,024. This, in combination with the lower memory capacity on A64FX, implies larger potential coverage of the virtual address space and thus, higher memory access performance. On the other hand, it also has implications on the cost of TLB invalidation, which we will detail in Section 4.2.2.

4 PERFORMANCE SCALING LINUX ON FUGAKU

4.1 Overall Approach for Optimizing Application Performance

Our overall approach is to avoid custom modifications to the Linux kernel source code shipped by RHEL as much as possible. This decision has been partly driven by our experience operating the K Computer, where custom kernel changes prevented us from following the mainline kernel's development due to maintainability issues [28]. When a fix is absolutely necessary to the Linux kernel, we promote patches directly to the RHEL distribution and do not apply them independently.

Fugaku provides a scalable execution environment for high-performance computing applications. To provide such an environment, system operation overhead must be reduced. Specifically, we use the following techniques for this purpose.

4.1.1 Containerization. On Fugaku, all applications run in Docker containers. Users can either use container images defined by the system administrators or their job will execute in a mode that appears to run on the host Linux. Such jobs have direct access to the host's root file system. In either case, Docker creates `cgroups` under the hood to manage compute resources for containers. An application `cgroup` is used to limit application memory usage and to bind user processes to specific cores and non-uniform memory access (NUMA) domains. In addition, we create a dedicated `cgroup` for system processes to isolate system CPUs and memory, which we detail in Section 4.2.

4.1.2 Virtual NUMA nodes. On A64FX, it was introduced into the system firmware that the physical address space are divided into system and application areas which are exposed as different NUMA domains to the Linux kernel. We call this technique virtual NUMA nodes. Memory fragmentation is an important factor that affects performance, which occurs with newly allocated memory areas that have been used before. Such allocations can consequently degrade performance for future memory allocations. The virtual NUMA node technique ensures that memory allocations by non-application processes can not utilize application dedicated memory areas, thus mitigating performance degradation by memory fragmentation.

4.1.3 Large page support. For applications that use large amounts of memory, the cost of virtual memory translations (*i.e.*, TLB misses and HW page table walks) affects execution performance. We prioritize larger page (a.k.a., huge page) backed memory allocations over the normal page size. Huge pages reduce the cost of the address translation process and improve memory access performance.

ARM64 supports multiple base page sizes [2]. On a typical `aarch64` system the base page size is 4KB, but RHEL is using a base page size of 64KB. ARM64 also provides a special feature called the page table *contiguous bit*, which enables the virtual to physical address translation for 32 physically contiguous pages to be cached by a single TLB entry if a designated bit in each page table entry is set, which can significantly reduce TLB misses.

Linux supports two methods for utilizing large pages, Transparent Huge Pages (THP) and hugeTLBfs. With 64kB base page size, using the contiguous bit results in 2MB sized large pages, while the regular large page is 512MB. Unfortunately, 512MB sized large pages easily lead to memory fragmentation problems and to inefficient utilization of large pages in general. As Linux only supports the contiguous bit feature in hugeTLBfs and not in the THP implementation, we opted to use hugeTLBfs on Fugaku. Normally, hugeTLBfs reserves a pool of large pages at boot time to ensure their availability, which in turn limits the number of normal pages available in the system. This can be a disadvantage for applications which who do not require large pages, *e.g.*, ones that do a lot of small dynamic allocations. To address both needs, we enable hugeTLBfs overcommit without reserving a pool and allocate large pages by the buddy allocator at runtime. However, the memory `cgroup` is not sufficiently integrated with hugeTLBfs and is unable to limit the usage of surplus large pages allocated by overcommit. To solve this problem, we hook a Linux kernel function in the cgroup implementation via a kernel module to override the default behavior and properly charge surplus hugeTLBfs pages to the memory `cgroup`.

Fugaku's runtime system provides a strong integration with the Linux kernel's hugeTLBfs facility. It enables the usage of large page backed memory for all process memory areas, such as static data (*i.e.*, `.data` and `.bss`), the stack area as well as the heap (the dynamic memory area primarily managed through the `mmap()` system call). The allocation scheme (*i.e.*, pre-allocation based or demand paging) can be controlled by specific environment variables.

4.1.4 NUMA aware process and thread binding. The A64FX node topology is organized around four application NUMA domains with 12 CPU cores on each. To maximize data locality, Fugaku's job scheduler automatically binds MPI process to specific NUMA domains depending on the number of ranks per node and users do not need to deal with the intricate interfaces of process binding provided by MPI implementations. We note that sophisticated users may choose to disable the default binding behavior and use the standard Linux interfaces (*e.g.*, `numactl`) if they wish so.

4.1.5 Hardware Barrier. The A64FX hardware provides a synchronization method for parallel applications called the hardware barrier [12] to accelerate synchronization between processes and/or threads inside a node. Support for the hardware barrier feature is integrated into the Fugaku's runtime system, *e.g.*, into the OpenMP implementation.

4.2 OS Noise Elimination Techniques

As mentioned in Section 2, in large scale supercomputing systems OS noise can have a significant impact on application performance. We worked together with RedHat to reduce OS noise in order to take advantage of having a standard distribution and to avoid custom changes to the Linux kernel. We describe our methodology in this section. Much of the OS noise is eliminated by dividing the system resources into a system CPU core partition and an application CPU core partition and binding system tasks to the former partition while application tasks to the latter. Specifically, the following hardware resources are partitioned into system versus application slices.

CPU cores The A64FX processor has 2-4 assistant cores in addition to the 48 application cores. System processes such as

OS daemons are bound to system cores and application processes are bound to application cores by using `cgroups`. Device IRQs are routed to assistant cores by configuring the relevant `procfs` files (*e.g.*, `/proc/irq/IRQ_NUMBER/smp_affnity`). Additionally, `kworker` tasks are also bound to assistant cores by changing the CPU affinity value through their `sysfs` interface.

Memory The physical memory address space is divided into a system region and an application region using virtual NUMA nodes. System processes are bound to the system region and application processes are bound to the application region through the Linux kernel's `cgroup` facility.

CPU caches Cache blocks are also divided into a system segment and an application segment by using A64FX's dedicated feature to partition cache blocks, called *sector cache* [12]. Assistant cores are bound to the system region and application cores are bound to the application region by using this custom HW feature.

The strict partitioning of hardware resources has a profound impact on mitigating interference between the operating system's internal activities and the application processes. However, various software interference remains. We discuss how such jitter can be mitigated focusing specifically on kernel and user level components.

4.2.1 Techniques in Kernel Space. For identifying kernel mode tasks that interfere with application code we utilize execution time profiling and `ftrace`, the Linux kernel's call tracing facility. Such interfering tasks can be suppressed by disabling them entirely, reducing their invocation frequency and/or binding them to assistant CPU cores. For example, the `ftrace` analysis revealed that a kernel thread for block I/O processing is spawned to application cores when using `blk-mq` even when unbound `kworker` tasks are bound to assistant cores because their binding is controlled by a dedicated data structure in the kernel (*i.e.*, `struct blk_mq_hw_ctx.cpumask`). In order to force them to specific processor cores we explicitly update the aforementioned CPU mask.

Another notable issue was a periodic access to performance monitoring unit (PMU) counters, such as the ones obtained with the `perf_event_open()` system call. We identified that PMU counters were read on all CPU cores in kernel space (involving IPIs), even if the access was initiated by a process bound to an assistant CPU core. After careful investigation, we found that the Fujitsu technical computing suite (TCS) [11], a middleware product that provides exascale system operation and application development environments, initiated these PMU access operations. TCS job operation software collects PMU counters to obtain number of execution cycles, floating-point instruction operations, memory read requests, memory write requests, and sleep cycles. We resolved this problem by providing a command that allows users to stop the automatic reading of PMU counters on a per-job basis, thus eliminating the implied interference.

4.2.2 Techniques in User Space. OS noise usually refers to the delay in the execution of application code due to activities in the OS kernel, such as an interrupt handler. However, that is not all. In today's multi-core systems, the OS code running on one CPU core may affect the execution time of applications running on another processor core. The occurrence of such noise can be confirmed by capturing the number of instructions retired and the execution time in user space and kernel space, respectively, using the performance counters of the CPU. If the number of executed instructions in kernel space increased, one can attribute such noise to OS processing, such as to interrupt or page fault handlers. On the other hand, when the execution time increases due to hardware sharing or internal contention in the hardware, there is no change in the number of executed instructions neither in user space nor in kernel space, and only the execution time increases.

Such interference may occur due to the fact that memory bandwidth to the main memory and/or to the last level cache are shared by multiple CPU cores. In Fugaku, TLB flush processing also had a large effect on performance, so countermeasures were required. As opposed to sending IPIs to specific processor cores for explicit remote TLB invalidation, the ARM64 ISA provides a special option to the TLB flush instruction (*i.e.*, TLBI) so that invalidation is performed in the entire Inner-Sharable domain, which entails all CPU cores on the chip. On A64FX used by Fugaku, we found that the execution of this instruction affects the performance of other CPU cores due to the relatively large TLB caches of A64FX (see Table 1). Experiments have confirmed that a delay of about 200 ns is generated by a single TLB flush instruction in A64FX. On Linux, some operations that release large amounts of memory, such as garbage collection at Go's runtime system and process termination operations, can cause hundreds to thousands consecutive TLB flushes, resulting in hundreds of microseconds of noise.

We worked with RedHat to address this issue by upstreaming Linux changes to reduce TLB flush broadcasts [1] and by incorporating improvements into RHEL 8.2 for use specifically for Fugaku's operation. In particular, the patch reduces TLB flush processing by utilizing a TLB flush instruction that affects only one core without broadcasting TLB flushes for processes that have all threads on a single CPU core, such as single-threaded processes. As with other ISAs (*e.g.*, x86_64 and SPARC64), it is also possible to implement all TLB flush processing in software that combines IPI and local TLB flush. However, Arm64 is originally designed with a relatively fast hardware implementation so that TLB flush processing to all processor cores can be called in one instruction and software implementation of multi-core TLB flush processing is significantly slower than the hardware implementation. Therefore, only the unnecessary process of broadcasting TLB flush for processes executed on a single CPU core is reduced and we continue using the broadcast based instruction for flushing TLBs on multiple CPU cores. In TCS, all components necessary for system operation, such as OS daemons and interrupts are bound to a single specific processor core.

5 IHK/MCKERNEL

This section gives a general overview of IHK/McKernel and describes recent developments targeted for the Fugaku system.

The IHK/McKernel multi-kernel operating system is comprised of two main components. A low-level software infrastructure, called Interface for Heterogeneous Kernels (IHK) [47], provides capabilities for partitioning resources in a many-core environment (*e.g.*, CPU cores and physical memory) and it enables management of

lightweight kernels. McKernel is a lightweight co-kernel developed on top of IHK. An overview of the multi-kernel architecture is depicted in Figure 2.

IHK is capable of allocating and releasing host resources dynamically and no reboot of the host machine is required when altering configuration. It is implemented as a collection of Linux kernel modules without any modifications to the Linux kernel itself, which enables straightforward deployment of the multi-kernel stack on a wide range of Linux distributions. Besides resource and LWK management, IHK also facilitates an Inter-kernel Communication (IKC) layer, which is used for implementing system call delegation (discussed below).

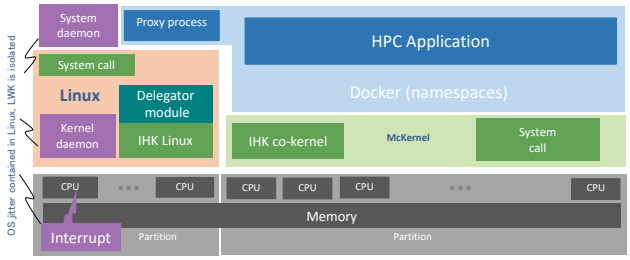


Figure 2: Architectural overview of IHK/McKernel and its integration with Docker containers

McKernel has been developed from scratch and while it is designed explicitly for high-performance computing workloads it retains a Linux compatible application binary interface (ABI) so that it can execute unmodified Linux binaries. There is no need for recompiling applications or for any McKernel specific libraries. McKernel implements only a small set of performance sensitive system calls and the rest of the OS services are delegated to Linux. Specifically, McKernel implements memory management, it supports processes and multi-threading, it has a simple round-robin co-operative (tick-less) scheduler, and it supports standard POSIX signaling. It also implements inter-process memory mappings and it offers interfaces for accessing hardware performance counters.

For each OS process executed on McKernel there is a process running on Linux, which we call the *proxy-process*. The proxy process' main role is to assist system call offloading. Essentially, it provides the execution context on behalf of the application so that offloaded system calls can be invoked in Linux. For more information on system call offloading, refer to [17]. The proxy process also provides means for Linux to maintain various state information that would have to be otherwise kept track of in the co-kernel. McKernel for instance has no notion of file descriptors, but it simply returns the number it receives from the proxy process during the execution of an `open()` system call. The actual set of open files (*i.e.*, file descriptor table, file positions, etc.) are managed by the Linux kernel. Relying on the proxy process, McKernel provides transparent access to Linux device drivers not only in the form of offloaded system calls (*e.g.*, through `write()` or `ioctl()`), but also via direct device mappings. Details of the device mapping mechanism has been described elsewhere [18].

5.1 Development for Fugaku

IHK/McKernel was originally developed for `x86` [18, 47]. Part of the Fugaku effort has been to port the multi-kernel OS to Fugaku's hardware, *i.e.*, supporting the `aarch64` ISA with all the ARM specific system level requirements. Additionally, we have integrated IHK/McKernel into Fugaku's containerized runtime as well as into the batch job submission system. Fugaku runs a proprietary job scheduler developed by Fujitsu. As opposed to OFP, where booting IHK/McKernel entails nothing more than calling a few privileged mode scripts in the prologue and epilogue of a particular job, on Fugaku there is a much tighter integration between IHK/McKernel and the Fujitsu environment. This is primarily due to the unique features of the Fugaku platform (*e.g.*, the hardware barrier, the way how process placement is performed, its interaction with MPI, etc., which we described in Section 4.1). One may consider the LWK as a plugin replacement to the `cgroup` facility of the Linux kernel with the important addition of its ability of kernel level specialization. In combination with containers, which enable customization of user-space components, the multi-kernel plus container approach enables specialization of the entire software stack [15].

Finally, another notable extension to McKernel is the Tofu PicoDriver. At high level, the Tofu network's system programming interface provides similar abstractions to that of Infiniband or Intel's OmniPath, but at the implementation level there are many subtle differences. For example, the registration of the so called STAGs, a concept similar to the Infiniband `verbs` layer's memory registration is performed through `ioctl()` calls into the Tofu driver. Because by default this is offloaded to Linux in our multi-kernel framework, it introduces additional latency. To eliminate such overhead we have developed a similar split driver infrastructure to that of OmniPath's PicoDriver described in [16]. We note that all of our experiments have been conducted using this capability.

6 EVALUATION

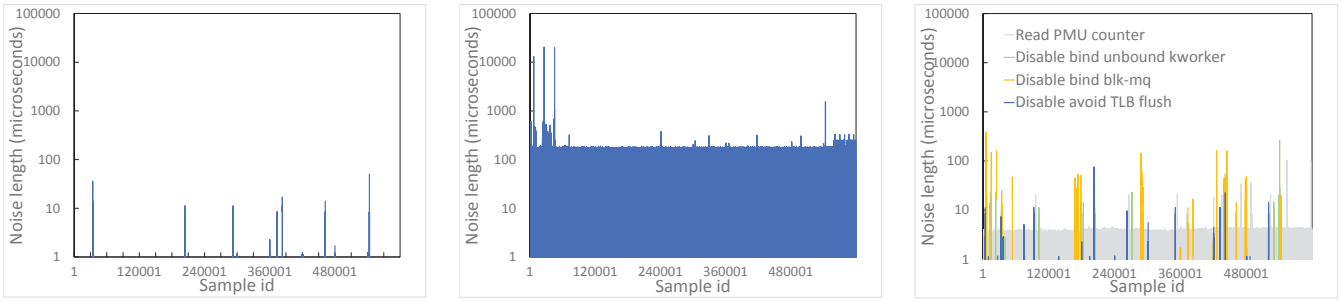
This section compares the performance of IHK/McKernel with Linux on two supercomputers. We describe the platforms and the benchmarks we used and present experimental results.

6.1 Experimental Environment

In addition to the platform description in Section 3 we provide further details of the exact software environment used in this paper. For all of our experiments on OFP, we configured the KNL processor in Quadrant flat mode; *i.e.*, MCDRAM and DDR4 RAM are addressable at different physical memory locations and appear as different NUMA domains. Applications were compiled with Intel compiler 19.0.3.199 and use the Intel MPI Version 2019 Update 3 Build 20190214 environment. For the McKernel measurements we deployed IHK and McKernel, commit hash `3bd05` and `da77a`, respectively. We utilized IHK's resource partitioning feature to reserve processor cores and physical memory dynamically.

On Fugaku, we compare the Linux environment described in Section 3 and Section 4 with IHK and McKernel commit hash `797a2` and `b9edb`, respectively. TCS version used for our experiments is 1.2.30a, a version used during the early access program of Fugaku.

IHK/McKernel is open source and publicly available at: https://github.com/RIKEN-SysSoft/mckernel.



(a) All countermeasures enabled **(b) Daemon binding disabled** **(c) Individual countermeasures disabled**

Figure 3: Impact of individual noise countermeasures on Fugaku Linux as captured by FWQ

6.2 Benchmarks and Applications

For noise measurement we use the Fixed Work Quanta (FWQ) benchmark [32]. FWQ performs a fixed amount of work in a loop, which contains only computation and does not access memory nor performs file I/O, it records the execution time for each loop iteration. System noise can be measured by the difference in the values. In all of our FWQ measurements we configure the benchmark to run for approximately 6.5ms, the largest value we could configure below 10ms on Fugaku. OFP uses the same target interval in terms of elapsed time. We chose this interval to match Linux' default timer interrupt frequency.

As for application level evaluation, we use the following codes.

- **AMG2013** is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. The code is written in ISO standard C [21].
- **Milc** represents part of a set of codes written by the MIMD Lattice Computation (MILC) collaboration used to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics. It performs simulations of four dimensional SU(3) lattice gauge theory on MIMD parallel machines. Strong interactions are responsible for binding quarks into protons and neutrons and holding them all together in the atomic nucleus [35].
- **Lulesh** is the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics benchmark, which is part of the Shock Hydrodynamics Challenge Problem. It was originally defined and implemented by LLNL as one of five challenge problems in the DARPA UHPC program and has since become a widely studied proxy application in DOE co-design efforts for exascale [27].
- **LQCD** benchmarks the performance of a linear equation solver with a large sparse coefficient matrix appearing in lattice Quantum Chromodynamics (QCD) simulations explaining the nature of protons and neutrons in terms of elementary particles called quarks and gluons. The four dimensional (space and time) coordinate is latticized and the equation of motion for quarks is converted to a large scale linear equation by the finite-difference method. It solves the equation for the O(a)-improved Wilson-Dirac quarks using the BiCGStab algorithm [25].
- **GeoFEM** solves 3D linear elasticity problems in simple cube geometries by parallel finite-element method (FEM). Trilinear hexahedral elements are used for the discretization. The Conjugate Gradient solver preconditioned by Incomplete Cholesky Factorization (ICCG) is applied for solving linear equations with sparse coefficient matrices. Additive Schwartz Domain Decomposition is introduced for stabilization of the parallel preconditioner [34].
- **GAMERA** solves 3D nonlinear seismic wave propagation problems in complex geometry domain based on implicit low-order finite-element method. Second-order tetrahedral elements are used for discretization, with a multi-grid and mixed precision arithmetic enhanced adaptive conjugate gradient solver. A matrix free matrix-vector multiplication method is used for reducing memory transfer and footprint [23].

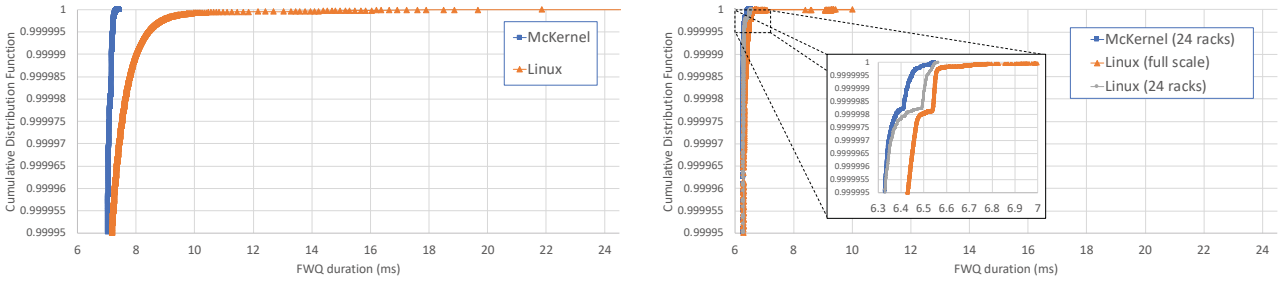
AMG2013, Milc and Lulesh are from the CORAL benchmark suite for which we have only `x86_64` optimized versions. As A64FX optimized versions of these codes are not available, we provide results on these applications using only OFP.

On the other hand, LQCD and GAMERA are two of the priority target applications of the Fugaku development project. Both of these have highly optimized versions for both target platforms that entail substantial code changes, but the different versions of the applications address the same science problem. As for GeoFEM, while it has a highly optimized version for OFP, it also has a few minor tweaks to support efficient execution on Fugaku.

6.3 Results of OS Noise Elimination Techniques on Linux

This section provides evaluation of the impact of various noise elimination techniques on Linux. We emphasize that these techniques have been only applied to Fugaku, as we have no control over the OFP Linux environment, we simply report the results we captured on that platform.

The noise elimination techniques are evaluated by using FWQ. The metrics used are the rate of noise that occurs per unit time (which we call noise rate) and the maximum noise length. The noise impact on bulk-synchronous applications is often dominated by the ratio of the maximum noise length to the synchronization interval, which has been shown in the past through simulations as well as



(a) FWQ latency CDF @ Oakforest-PACS

(b) FWQ latency CDF @ Fugaku

Figure 4: FWQ latency cumulative distribution function on OFP and Fugaku, comparing Linux vs. McKernel

kernel level noise injection [10, 22]. Using Eq. 1, this is because the $1-(1-\frac{S}{I_i})^N$ component, which represents the probability for the max noise to hit and delay at least one of the N synchronization intervals gets close to 1 due to the large N. For example on full scale Fugaku, where N equals to 7,630,848 (the total number of HW threads), even with as small rate as once in every 600 seconds the probability is close to 1.

Table 2: Effectiveness of individual noise elimination technique

Disabled technique	Maximum noise length (us)	Noise rate
None	50.44	3.79E-6
Daemon process	20346.98	9.94E-4
Unbound `kworker` tasks	266.34	4.58E-6
`blk-mq` worker tasks	387.91	4.58E-6
PMU counter reads	103.09	8.27E-6
CPU-global flush instruction	90.2	3.87E-6

The noise elimination techniques we evaluate are as follows. Binding daemon processes to assistant cores, binding unbound `kworker` tasks to assistant cores, binding `blk-mq` worker tasks to assistant cores, stopping periodic PMU counter reads, and suppressing CPU-global TLB flush instruction. For each of these measures, we capture FWQ results with the selected one turned off and compare it with a baseline where all countermeasures are enabled.

We note that these measurements were performed on an in-house 16-node A64FX system with identical hardware and software environment to that of the main Fugaku system. Table 2 shows the maximum noise lengths and the corresponding noise rates, calculated as follows. Let us denote the execution time of i-th FWQ loop by T_i, T_{max} and T_{min} then indicate the maximum and minimum values across all loops, respectively. The maximum noise length is calculated by $T_{max} - T_{min}$ and the noise rate is obtained by the following formula:

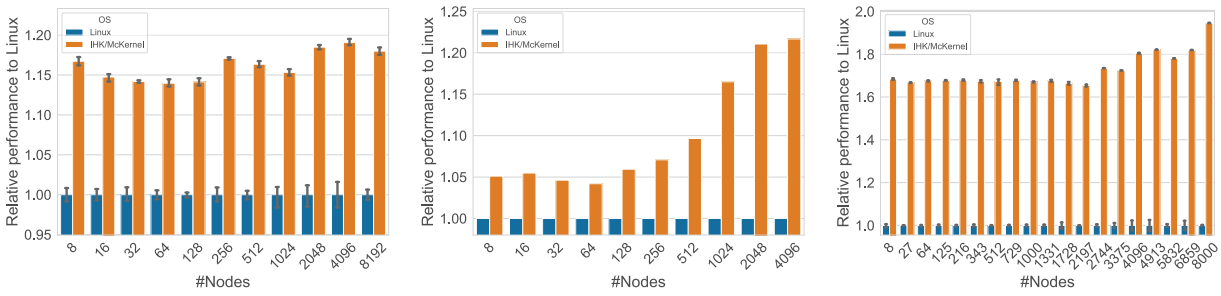
$$\frac{\sum_{i=0}^{n} \frac{T_i - T_{min}}{T_{min}}}{n} \tag{2}$$

In addition, we plot time series of noise length data calculated from the output of FWQ. Noise length L_i is calculated as $T_i - T_{min}$. Figure 3a shows the noise lengths when FWQ runs with all countermeasures enabled, while Figure 3b and Figure 3c show the noise lengths with individual countermeasures disabled. The X axis indicates sample ID, *i.e.*, the i-th loop of FWQ and the Y axis plots the corresponding L_i. Sample ID increases once in every approximately 6ms in an ideal condition without noise.

As one can see from the table and the figures, each measure eliminates a significant amount of jitter from the system. Making sure OS daemons are restricted to assistant cores has the most pronounced effect, eliminating as much excess noise as 20 milliseconds. Binding `kworkers`, `blk-mq` and the avoidance of global TLB invalidations have an impact in the range of up to 400μs. However, there still remains some OS noise even when applying all of the noise elimination techniques. We found that the main cause of the remaining noise is a Linux tool called `sar`, which periodically monitors system activities including CPU, memory, I/O, network, context switches and paging. This service is required on Fugaku to be turned on for operation purposes.

To get a better assessment of OS noise in the overall system we also captured FWQ data at scale. Specifically, we extended FWQ to run on an arbitrary number of nodes (using MPI) and measure OS noise on all CPU cores simultaneously. We ran ten iterations of measurements that last for approximately 6 minutes, capturing a noise profile that covers one hour altogether. Due to the large amount of raw data that would need to be saved, we in-situ select the 100 worst performing compute nodes (*i.e.*, the ones with the largest noise duration) and save data to the parallel file system only on those. Using all the data captured we plot cumulative distribution functions of the noise data in Figure 4 comparing OFP against Fugaku, using both Linux and IHK/McKernel.

On OFP, we used 1,024 nodes for this experiment because we did not have exclusive access to the entire machine. On Fugaku, we provide data for the following three configurations. For McKernel, we had exclusive access to 24 racks (9,216 compute nodes) and provide data on that scale. For Linux, we show measurements using the full scale (158,976 compute nodes) Fugaku system, and for fairness we also use the same 24 racks on which the McKernel results were obtained. X axis indicates the FWQ iteration length and Y axis represents the cumulative distribution function (*i.e.*, the tail latency of FWQ). We scale the X axis on both OFP (Figure 4a) and Fugaku



(a) AMG2013 (b) Milc (c) Lulesh

Figure 5: Application results exclusively on Oakforest-PACS

(Figure 4b) results to the same interval so that results from the two systems can be easily compared.

The first observation is that OFP is significantly more jittery compared to Fugaku. In particular, on OFP we observe FWQ iterations under Linux that last for up to 24ms (note that the benchmark is configured to capture periods of 6.5ms), while on Fugaku the largest FWQ value is around 10ms. With respect to Linux vs. IHK/McKernel, on OFP McKernel provides significant noise reduction, the largest value on McKernel remains smaller than 7ms. However, the situation on Fugaku is more intricate. While the full scale Linux numbers clearly look more jittery than McKernel, Linux on 24 racks is in fact not that different, only slightly worse than McKernel's performance. Due to resource limitations and stability issues we were unable to capture IHK/McKernel results on full scale, but knowing that it performs absolutely no background activities we speculate that noise profile on full scale would not look significantly different than on 24 racks.

6.4 Application Level Results

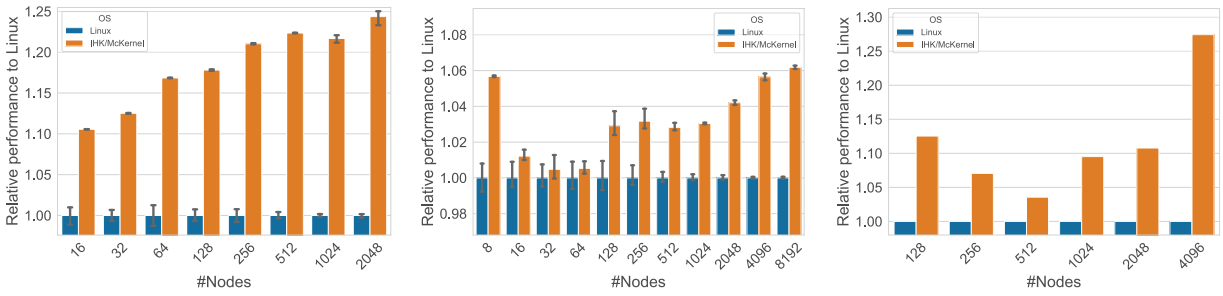
Let us turn our attention to application level evaluation. As we described in Section 6.2, we gathered results on three CORAL mini-apps using only OFP and another three primarily from the Fugaku development project comparing Linux with IHK/McKernel both on OFP and on Fugaku. We note that similarly to the FWQ measurements, on Fugaku resource limitations and stability problems prevented us from gathering full scale measurements and we present numbers on up to 24 racks. On both platforms we utilize IHK/McKernel's integration with the job submission system and run the measurements through the batch job system, however, we note that on Fugaku we made sure that for each node count the exact same compute nodes are utilized for both the Linux and McKernel measurements. We also note that while we show numbers on both platforms using up to 8k compute nodes, because the number of HW threads on the Xeon Phi chip is significantly larger than that on A64FX, in absolute terms OFP results represent higher level of parallelism, 2,097,152 and 393,216 HW threads on OFP and Fugaku, respectively.

Figure 5 shows application level results for the CORAL workloads on OFP. On each plot, X axis denotes the number of compute nodes and Y axis indicates performance. Linux numbers (blue bars) are normalized to one for each node count and McKernel performance (orange bars) indicate relative performance to Linux. We plot relative performance as opposed to runtime because some applications report custom metrics. Unless stated otherwise we ran each experiment at least three times and show mean performance with error bars indicating variation from the mean where enough data are available.

As seen in Figure 5, IHK/McKernel outperforms Linux on all CORAL benchmarks we used for evaluation. On AMG2013 we observe up to approximately 18% performance improvement with a slight tendency of increasing advantage as we scale out. On the other hand, we observe a more pronounced performance increase with the growing scale when running Milc and Lulesh, for which McKernel attains up to 22% and almost 2X improvement, respectively. As we discussed in previous work, the improvement of Lulesh mainly stems from heap management issues in Linux [14].

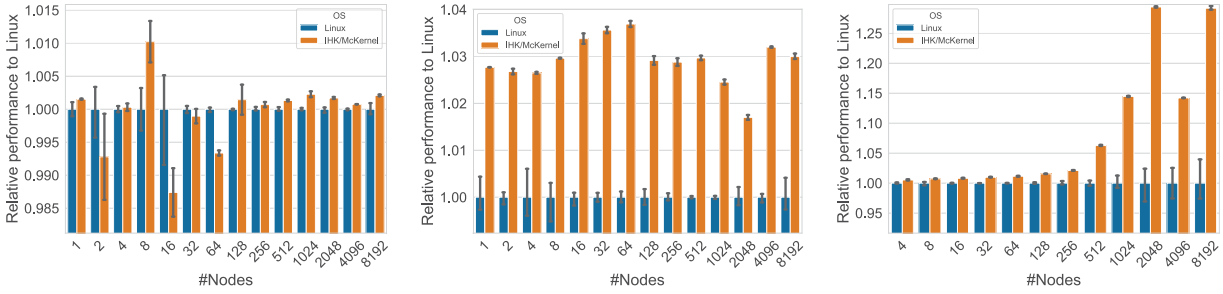
Figure 6 and Figure 7 summarize application level results comparing OFP and Fugaku on LQCD, GeoFEM and GAMERA. Similarly to the CORAL applications, McKernel on OFP consistently outperforms Linux on these applications as well. Results are shown in Figure 6. Although we have results for only up to 2k nodes on LQCD, we observe an increase in McKernel's performance gain as we scale to larger node counts reaching close to 25% at 2k nodes. For GeoFEM, we have full-scale OFP measurements with McKernel outperforming Linux by up to 6% at the entire machine. To much of our surprise we also find a significant amount of variation (indicated by large error bars) across measurements even on McKernel, although we believe this could be related to the fact that different measurements run on different nodes, which is particularly true for smaller node counts. With respect to GAMERA, we have limited data points for showing error bars on all scales and decided to show only the mean performance. As seen, McKernel outperforms Linux by over 25% on half-scale of the OFP supercomputer.

Fugaku results are shown in Figure 7. We observe substantially lower performance gain with McKernel over Linux on the Fugaku machine, although arguably 8k compute nodes on Fugaku is still relatively small scale when considering the overall system's node count. LQCD performs almost identical on the two operating systems. For GeoFEM, we observe an average of 3% performance improvement over Linux when running on McKernel and the tendency indicates the gain would possibly remain constant even if we scaled



(a) LQCD (b) GeoFEM (c) GAMERA

Figure 6: Application results on Oakforest-PACS



(a) LQCD (b) GeoFEM (c) GAMERA

Figure 7: Application results on Fugaku

further out. Only for GAMERA we see an increase in performance gain reaching up to 29% on McKernel when running on 8k compute nodes. We did investigate the GAMERA results further and found that McKernel performs significantly better in the first step (out of three) of the application, which may be related to some sort of overhead in initialization on Linux. Had the application run further steps we would likely see McKernel's advantage decrease. Unfortunately, we had a very narrow window of opportunity for pinpointing where the performance difference stems from, but we did observe faster RDMA registration in McKernel due to the LWK integrated Tofu driver (described in Section 5.1), which we suspect as one of the main contributors to the performance improvement.

7 RELATED WORK

Without striving for completeness, this section discusses the most related studies in the domain of operating systems for HPC. A more complete coverage of this domain is presented in [13].

Lightweight kernels (LWKs) [43] designed for HPC applications date back to the early 1990s. These kernels ensure low operating system noise, high scalability and predictable performance for large scale scientific applications. Catamount [29], developed at Sandia National laboratories, was one of the first LWKs that has been deployed in a production environment. IBM's BlueGene line of supercomputers have also been running an HPC specific LWK called the Compute Node Kernel (CNK) [20]. While Catamount has been written entirely from scratch, CNK builds upon a substantial amount of code from Linux so that it could provide better support standard UNIX features. The most recent LWK from Sandia National Laboratories is Kitten [39], which is unique compared to the prior LWKs because it provides a more complete Linux-compatible environment. There are also LWKs that start from a full Linux system and modifications are introduced to meet HPC requirements. Cray's Extreme Scale Linux [36, 41], ZeptoOS [51] and the OS that was deployed on the K Computer [28] follow this path. These efforts usually eliminate daemon processes, simplify the scheduler, and replace the memory management system. Linux' complex code base, however, can make it difficult to mitigate all the undesired effects, as we also showed in this paper when capturing FWQ numbers on the full scale Fugaku machine. In addition, for projects that do modify the Linux kernel it is also cumbersome to keep those modifications in sync with the rapidly evolving Linux source code.

With the advent of many-core processors, the lightweight multi-kernel approach has been proposed [18, 30, 37, 50]. Multi-kernels run Linux and an LWK side-by-side on different cores of the CPU to provide OS services in collaboration between the two kernels. FusedOS [38] was the first proposal that pioneered this approach. It's primary motivation was to address CPU core heterogeneity between system and application cores. In contrast to McKernel, FusedOS runs the LWK at user level. The kernel code on application CPUs in the FusedOS prototype 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. The FusedOS work was the first to demonstrate that Linux noise can be completely isolated to the Linux cores and avoid interference with HPC applications running on the LWK cores. This attribute of the multi-kernel approach has been also one of the driving forces for the McKernel model.

From more recent multi-kernel efforts, one of the most similar projects to ours is Intel's mOS [19, 50]. mOS follows a path of much stronger integration with Linux and can directly take advantage of some of the Linux kernel infrastructure. Nevertheless, this approach comes at the price of Linux modifications and an increased complexity in eliminating OS interference.

Argo [5] is an exa-scale OS project targeted at workflow like applications. The Argo approach is using OS and runtime specialization (through enhanced Linux containers) on compute nodes, to some extent similar to Fugaku's usage of container technologies. Argo expects to use a ServiceOS based on Linux to boot the node and run management services. It then runs different container instances that cater to the specific needs of applications or pieces of a workflow.

Hobbes [6] is a DOE Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is improved support for application composition. Hobbes utilizes virtualization technologies to provide the flexibility to support the requirements of application components for different node-level OSes. At the bottom of its software stack, Hobbes runs Kitten [39] as its LWK, on top of which Palacios [31] serves as a virtual machine monitor. As opposed to IHK/McKernel, Hobbes separates Linux and Kitten at the PCI device level, which implies some difficulties in supporting the full POSIX APIs and in porting necessary device drivers to Kitten.

8 CONCLUSION AND FUTURE WORK

The HPC OS community has been living with the presumption that lightweight kernels have the potential to outperform Linux at extreme scale scientific computing. Yet, no large scale studies have been performed where an LWK is compared with a highly tuned Linux environment.

In this paper, we have presented IHK/McKernel, a lightweight multi-kernel operating system designed for high-performance computing. We deployed McKernel on two large scale HPC system, one of which is Fugaku, the fastest supercomputer on the TOP500 list at the time of writing this paper. We also described Fugaku's Linux software environment and the specific measures we took to scale it for HPC. Through rigorous evaluation using microbenchmarks as well as various HPC applications, we conclude that although our LWK can outperform Linux at extreme scale, a highly tuned Linux environment provides close to LWK level performance, in the proximity of 4% on average across all experiments. However, according to our most recent experience on Fugaku, maintaining Linux scalability across OS updates (in particular when updating the Linux kernel) is a non-trivial effort that requires thorough investigation of Linux internals. As no such investigation is required for an LWK, we see this as a clear advantage of IHK/McKernel.

In the future, we will pursue further opportunities to improve the performance of our OS and we intend to evaluate it on a broader set of applications using the full scale Fugaku machine. We also believe that the multi-kernel operating system structure has a role to play in a future of heterogeneous processing elements (PE) as specialized PEs get increasingly capable and will eventually allow running privileged mode code. Finally, as it has been demonstrated before [37], multi-kernel systems provide excellent performance isolation which could play an important role in multi-tenant deployments on accelerator equipped fat compute nodes, a direction we also consider for future investigation.

ACKNOWLEDGMENT

This work has been funded by the Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT) program for the Development and Improvement of the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities.

We thank the operation teams of both OFP and Fugaku for their relentless support in the deployment and operation of IHK/-McKernel. In particular, we are grateful to Yoshio Sakaguchi, Toshirou Saiki, Atsushi Ninomiya and Fumichika Sueyasu from Fujitsu. We are also grateful to the Fujitsu TCS developer team for their help with noise reduction and to RedHat for helping to upstream Linux kernel changes and for incorporating improvements into RHEL.

We would also like to thank the mOS team of Intel Corporation for the invaluable discussions on LWK design and implementation as well as for their help with the CORAL mini-applications.

REFERENCES

[1] Andrea Arcangeli. 2020. Linux patch: Arm64: TLB: skip TLBI broadcast v2. https://lkml.org/lkml/2020/2/23/188.

[2] ARM. 2021. Arm Architecture Reference Manual Armv8. https://developer.arm.com/documentation/ddi0487/latest/

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 66, 11 pages.

[4] BDEC Committee. 2017. The BDEC "Pathways to Convergence" Report. http://www.exascale.org/bdec/.

[5] Pete Beckman, Marc Snir, Pavan Balaji, Franck Cappello, Rinku Gupta, Kamil Iskra, Swann Perarnau, Rajeev Thakur, and Kazutomo Yoshii. 2017. Argo: An Exascale Operating System. http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[6] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. 2013. 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* (Eugene, Oregon) *(ROSS)*.

[7] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. 2008. SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-core Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas) *(SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 25, 12 pages.

[8] R. Brightwell, R. Riesen, K. Underwood, T. B. Hudson, P. Bridges, and A. B. Maccabe. 2003. A performance comparison of Linux and a lightweight kernel. In *2003 Proceedings IEEE International Conference on Cluster Computing*. 251–258.

[9] Tsafrir Dan, Etsion Yoav, G. Feitelson Dror, and Kirkpatrick Scott. 2005. System noise, OS clock ticks, and fine-grained parallel applications. In *19th annual international conference on Supercomputing*.

[10] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5219920

[11] Fujitsu Limited. 2019. White paper Advanced Software for the FUJITSU Supercomputer PRIMEHPC FX1000. https://www.fujitsu.com/downloads/SUPER/primehpc-fx1000-soft-en.pdf

[12] Fujitsu Limited. 2021. A64FX® Microarchitecture Manual. https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.4.pdf
[13] Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, and Robert W. Wisniewski (Eds.). 2019. *Operating Systems for Supercomputers and High Performance Computing*. Vol. 1. Springer. https://doi.org/10.1007/978-981-13-6624-6
[14] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Yutaka Ishikawa, and Robert W. Wisniewski. 2018. Performance and Scalability of Lightweight Multi-Kernel based Operating Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
[15] Balazs Gerofi, Rolf Riesen, Robert W. Wisniewski, and Yutaka Ishikawa. 2017. Toward Full Specialization of the HPC Software Stack: Reconciling Application Containers and Lightweight Multi-Kernels. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017* (Washingon, DC, USA) *(ROSS '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3095770.3095777
[16] Balazs Gerofi, Aram Santogidis, Dominique Martinet, and Yutaka Ishikawa. 2018. "PicoDriver: Fast-Path Device Drivers for Multi-Kernel Operating Systems". In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3208040.3208060
[17] Balazs Gerofi, Akio Shimada, Atsushi Hori, and Yutaka Ishikawa. 2013. 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)*.
[18] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Guo Nakamura, Tomoki Shirasawa, and Yutaka Ishikawa. 2016. 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)*.
[19] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W. Wisniewski. 2015. Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing. In *Proceedings of ROSS'15* (Portland, OR, USA). ACM, Article 5. https://doi.org/10.1145/2768405.2768410
[20] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. 2010. 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 (SC)*.
[21] V. E. Henson and U. M. Yang. 2002. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. https://codesign.llnl.gov/amg2013.php. *Appl. Num. Math.* 41 (2002), 155–177.
[22] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/SC.2010.12
[23] T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schulthess, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, M. Hori, and L. Maddegedara. 2018. A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 627–637. https://doi.org/10.1109/SC.2018.00052
[24] Intel Corporation. 2013. Intel Xeon Phi™ Core Micro-architecture. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-phi-core-micro-architecture.html
[25] Ken-Ichi Ishikawa, Yoshinobu Kuramashi, Akira Ukawa, and Taisuke Boku. 2017. CCS QCD Application. https://github.com/fiber-miniapp/ccs-qcd.
[26] Joint Center for Advanced HPC (JCAHPC). 2017. Basic Specification of Oakforest-PACS. http://jcahpc.jp/files/OFP-basic.pdf.
[27] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. Lawrence Livermore National Laboratory.
[28] Takeharu Kato and Kouichi Hirai. 2019. *K Computer*. Springer Singapore, Singapore, 183–197. https://doi.org/10.1007/978-981-13-6624-6_11
[29] Suzanne M. Kelly and Ron Brightwell. 2005. Software architecture of the light weight kernel, Catamount. In *Cray User Group*. 16–19.
[30] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. 2016. Decoupled: Low-Effort Noise-Free Execution on Commodity Systems. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2931088.2931095
[31] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Zheng Cui, Lei Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. 2010. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. https://doi.org/10.1109/IPDPS.2010.5470482
[32] Lawrence Livermore National Laboratory. [n.d.]. The FTQ/FWQ Benchmark. https://asc.llnl.gov/sequoia/benchmarks/FTQ_summary_v1.1.pdf
[33] J. Moreira, M. Brutman, J. Castano, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt. 2006. Designing a Highly-Scalable Operating System: The Blue Gene/L Story. In *SC 2006 Conference, Proceedings of the ACM/IEEE*. 53–53.
[34] Kengo Nakajima. 2003. 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* (Phoenix, AZ, USA) *(SC)*. ACM, New York, NY, USA. https://doi.org/10.1145/1048935.1050164
[35] NERSC. 2017. MILC. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/milc/.
[36] Sarp Oral, Feiyi Wang, David A. Dillow, Ross Miller, Galen M. Shipman, Don Maxwell, Dave Henseler, Jeff Becklehimer, and Jeff Larkin. 2010. Reducing Application Runtime Variability on Jaguar XT5. In *Proceedings of CUG'10*.
[37] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. 2015. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. ACM, New York, NY, USA, 149–160.
[38] Yoonho Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, Kyung Dong Ryu, and R.W. Wisniewski. 2012. 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*. 211–218. https://doi.org/10.1109/SBAC-PAD.2012.14
[39] Kevin T. Pedretti, Michael Levenhagen, Kurt Ferreira, Ron Brightwell, Suzanne Kelly, Patrick Bridges, and Trammell Hudson. 2010. *LDRD Final Report: A Lightweight Operating System for Multi-core Capability Class Supercomputers*. Technical report SAND2010-6232. Sandia National Laboratories.
[40] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 55.
[41] Howard Pritchard, Duncan Roweth, David Henseler, and Paul Cassella. 2012. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In *Proceedings of Cray User Group (CUG)*.
[42] Rolf Riesen, Ron Brightwell, Patrick G. Bridges, Trammell Hudson, Arthur B. Maccabe, Patrick M. Widener, and Kurt Ferreira. 2009. Designing and Implementing Lightweight Kernels for Capability Computing. *Concurrency and Computation: Practice and Experience* 21, 6 (April 2009). https://doi.org/10.1002/cpe.v21:6
[43] Rolf Riesen, Arthur Barney Maccabe, Balazs Gerofi, David N. Lombard, John Jack Lange, Kevin Pedretti, Kurt Ferreira, Mike Lang, Pardo Keppel, Robert W. Wisniewski, Ron Brightwell, Todd Inglett, Yoonho Park, and Yutaka Ishikawa. 2015. What is a Lightweight Kernel?. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (Portland, OR, USA) *(ROSS)*. ACM, New York, NY, USA. https://doi.org/10.1145/2768405.2768414
[44] RIKEN Center for Computational Science. 2021. Fugaku. https://www.r-ccs.riken.jp/en/fugaku/.
[45] Subhash Saini and Horst D. Simon. 1994. Applications Performance Under OSF/1 AD and SUNMOS on Intel Paragon XP/S-15. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (Washington, D.C.) *(Supercomputing '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 580–589.
[46] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu. 2020. "Co-Design for A64FX Manycore Processor and "Fugaku"". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00051
[47] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. 2014. 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 (HiPC)*.
[48] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181
[49] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. 1994. PUMA: an operating system for massively parallel systems. In *Proceedings of System Sciences'94*, Vol. 2. 56–65.
[50] Robert W. Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. 2014. mOS: An Architecture for Extreme-scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (Munich, Germany) *(ROSS)*. ACM, New York, NY, USA, Article 2.
[51] Kazutomo Yoshii, Kamil Iskra, Harish Naik, Pete Beckmanm, and P. Chris Broekema. 2009. Characterizing the Performance of Big Memory on Blue Gene Linux. In *Proceedings of the 2009 Intl. Conference on Parallel Processing Workshops (ICPPW)*. IEEE Computer Society, 65–72.
[52] J. A. Zounmevo, S. Perarnau, K. Iskra, K. Yoshii, R. Gioiosa, B. C. V. Essen, M. B. Gokhale, and E. A. Leon. 2015. A Container-Based Approach to OS Specialization for Exascale Computing. In *2015 IEEE International Conference on Cloud Engineering*. 359–364.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We use three applications in this paper, LQCD, GeoFEM and GAMERA running on two platforms and two operating systems each. LQCD is publicly available (see artifacts below), GeoFEM and GAMERA are not available online. For GeoFEM we got the source code directly from the developer (Prof. Nakajima Kengo @ The University of Tokyo). For GAMERA we used binary executables we received from the maintainer of the application (Dr. Kohei Fujita @ The University of Tokyo).

On Oakforest-PACS we used Intel compiler 19.0.3.199 and Intel MPI Version 2019 Update 3 Build 20190214. LQCD was run using 4 ranks per node and 32 OpenMP threads per rank. GeoFEM was run using 16 ranks per node and 8 OpenMP threads per rank. GAMERA was run with 8 ranks per node and 8 OpenMP threads per rank. For the Linux runs we utilize Intel MPI's process binding and the I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0-3,68-71,136-139,204-207 environment variable to exclude system CPU cores. On McKernel we use the -n mcexec option to automatically bind processes.

On Fugaku we used Fujitsu TCS 1.2.30a which contains Fujitsu MPI and Fujitsu OpenMP implementations with the same version. All applications were ran using 4 ranks per node and 12 threads per rank (i.e., one rank per A64FX CMG). The Fujitsu runtime binds MPI processes automatically.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪ https://github.com/fiber-miniapp/ccs-qcd
Artifact name: LQCD x86 version

Persistent ID: https://github.com/RIKEN-LQCD/qws
Artifact name: LQCD Fugaku optimized version

Persistent ID:
↪ https://github.com/RIKEN-SysSoft/mckernel
Artifact name: IHK/McKernel
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Oakforest-PACS: 8,192 compute nodes of Intel Xeon Phi 7250 Knights Landing CPU; Fugaku: 158,976 compute nodes of Fujitsu A64FX CPU

Operating systems and versions: Oakforest-PACS: CentOS 7.3, Linux kernel 3.10.0-693.11.6, IHK/McKernel: commit hash: 3bd05/da77a; Fugaku: RHEL 8.3 Linux kernel 4.18.0-240.8.1.el8_3, IHK/McKernel commit hash: 797a2/b9edb

Compilers and versions: Intel compiler 19.0.3.199, Fujitsu TCS 1.2.30a

Applications and versions: LQCD, GeoFEM and GAMERA

Libraries and versions: Intel MPI Version 2019 Update 3 Build 20190214 and Fujitsu MPI TCS v1.2.30a