PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems

Balazs Gerofi
RIKEN Advanced Institute for Computational Science
Japan
bgerofi@riken.jp

Aram Santogidis
CERN
Switzerland
aram.santogidis@cern.ch

Dominique Martinet
CEA
France
dominique.martinet@cea.fr

Yutaka Ishikawa
RIKEN Advanced Institute for Computational Science
Japan
yutaka.ishikawa@riken.jp

ABSTRACT

Lightweight kernel (LWK) operating systems (OS) in high-end supercomputing have a proven track record of excellent scalability. However, the lack of full Linux compatibility and limited availability of device drivers in LWKs have prohibited their wide-spread deployment. Multi-kernels, where an LWK is run side-by-side with Linux on many-core CPUs, have been proposed to address these shortcomings. In a multi-kernel system the LWK implements only performance critical kernel services and the rest of the OS functionality is offloaded to Linux. Access to device drivers is usually attained via offloading. Although high-performance interconnects are commonly driven from user-space, there are networks (e.g., Intel's OmniPath or Cray's Gemini) that require device driver interaction for a number of performance sensitive operations, which in turn can be adversely impacted by system call offloading.

In this paper, we propose *PicoDriver*, a novel device driver architecture for multi-kernels, where only a small part of the driver (i.e., the performance critical piece) is ported to the LWK and access to the rest remains transparent via Linux. Our solution requires no modifications to the original Linux driver, yet it enables optimization opportunities in the lightweight kernel. We implemented this system in the IHK/McKernel multi-kernel OS and demonstrate that on 16,384 Intel Knight's Landing (KNL) CPU cores (interconnected by OmniPath network) we can outperform Linux by up to 30% on various mini-applications.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; *Communications management*; Message passing; • **Networks** → *Network performance analysis*;

KEYWORDS

high-performance computing, operating systems, lightweight kernels, multi kernels, device drivers

ACM Reference Format:
Balazs Gerofi, Aram Santogidis, Dominique Martinet, and Yutaka Ishikawa. 2018. PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems. In *HPDC '18: International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3208040.3208060

1 INTRODUCTION

Lightweight kernel (LWK) [37] operating systems (OS) specifically designed for high-performance computing (HPC) workloads have been successfully deployed on a number of large-scale supercomputers. For example, Cougar [8] and Catamount [21], two LWKs that originate from the SUNMOS [38] and PUMA [41] OSes developed at Sandia National Laboratories and the University of New Mexico, were the default compute partition operating systems of the ASCI Red and Red Storm supercomputers, respectively. IBM's Compute Node Kernel (CNK) [17, 28] is another notable lightweight kernel that has been running on the BlueGene line of supercomputers with its latest reincarnation on the BG/Q still in production.

Lightweight kernels have a proven track record of excellent scalability, predictable performance, and the potential for providing a fertile ground for rapid experimentation with novel OS concepts due to their relatively simple code base [7, 8, 13, 30, 36]. However, the lack of device driver support in lightweight kernels and the limited compatibility with the standard POSIX/Linux APIs have prohibited their wide-spread deployment. Indeed, it has been reported that the main obstacle for carrying out large-scale evaluation of Kitten [27], the latest of the Sandia line of lightweight kernels, has been the lack of support for Infiniband networks [30]. At the same time, neither Catamount nor the IBM CNK provides full compatibility for a POSIX compliant `glibc`, limiting the availability of standard system calls, such as `fork()` [17]. Although the lack of full POSIX/Linux support has been increasingly becoming problematic with the recent shift to more diverse workloads in supercomputing environments (e.g., Big Data analytics and machine learning, etc. [3]), the unavailability of device drivers has been undoubtedly one of the major hindrances for LWKs' stronger dominance.

HPDC '18, June 11–15, 2018, Tempe, AZ, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208060

As a matter of fact, porting device drivers from one operating system to another can be a daunting task. While in essence the driver does nothing more than providing high level interfaces to the hardware, the underlying implementation of those functionalities can be substantially entangled with the internals of the specific OS. For example, a Linux device driver usually complies with the Linux device model, which provides facilities for device classes, hotplugging, power management and system shutdown, communication with user-space, and synchronization, etc. [10]. Linux device drivers implement file operations with callback functions registered to the Linux Virtual File System (VFS) layer and they often provide device specific entries in pseudo file systems such in `/proc` or `/sys`. Unless there is straightforward one-to-one mapping for these abstractions between the two operating systems, porting device drivers may demand significant development efforts. Needless to say, most of these kernel facilities do not exist in typical lightweight kernels. To some extent that is exactly one of the main merits of an LWK, i.e., keeping kernel internals simple and focusing on performance. Unfortunately, this renders porting a complete device driver to an LWK a major undertaking.

However, with the advent of multi-, and many-core CPUs, multi-kernel operating systems have been proposed to address the aforementioned shortcomings of LWKs [14, 16, 26, 30, 31, 42]. In a multi-kernel system, Linux and a lightweight kernel is run side-by-side on compute nodes with the motivation to provide LWK performance and scalability, to retain full compatibility with the Linux/POSIX APIs, and to reuse device drivers from Linux at the same time. The LWK component usually implements only a subset of OS services, i.e., the performance sensitive ones, and the Linux provided kernel facilities are attained via some form of co-operation between the two kernels. There have been multiple proposed solutions for interaction between the two kernels, for example using system call offloading or directly migrating execution contexts across kernel boundaries [16, 26]. While this inter-kernel communication (IKC) usually comes with additional overhead, the basic idea is that IKC takes place only in slow path operations and thus the extra overhead is irrelevant.

With respect to high-performance interconnects, multi-kernels excessively rely on the fact that most of the performance sensitive network operations are driven entirely from user-space, also known as OS bypass [2]. Therefore, involving Linux only at the time of device initialization or other infrequent administrative operations is a viable approach. Unfortunately, there are high-performance networks that do involve system calls (i.e., invocations that need to be offloaded to Linux) in performance sensitive operations as well. For example, both Intel's OmniPath [5] and Cray's Gemini [1] networks need device driver involvement for Remote Direct Memory Access (RDMA) transfers of large messages. Mellanox Infiniband [2] memory registration also requires system calls, although memory registration is not necessarily in the critical path of execution. These offloaded calls can adversely impact performance (occasionally to unacceptable levels) of parallel applications executing on top of a multi-kernel system.

As we already argued, porting the entire device driver to the LWK for each network device is impractical. In a multi-kernel system, however, one might take advantage of the fact that Linux with all of its device drivers is also present. Instead of porting the entire driver, we asked ourselves the following questions: Can we port only the performance critical part of the device driver to the LWK and keep utilizing Linux for the rest of the driver functionalities? Will such a driver architecture present any opportunities to further improve the fast-path code of the device driver by better matching it to LWK internals? Can we keep the Linux driver unmodified? In this paper, we embark on an exploration to answer exactly these questions. Specifically, we make the following contributions:

- We propose *PicoDriver*, a novel device driver framework that enables porting exclusively the performance critical part of a device driver to an LWK in a multi-kernel system while transparently retaining the rest of the driver functionality via Linux.
- We provide and implementation of the Intel OmniPath *PicoDriver* in the IHK/McKernel lightweight multi-kernel operating system [13–15].
- We demonstrate that our system requires no modifications to the original Linux driver, yet, it enables optimization opportunities in the LWK.
- We evaluate the proposed mechanism on 256 Intel Knight's Landing compute nodes (i.e., 16,384 KNL CPU cores) and show that McKernel can outperform Linux by up to 30% on various HPC mini-applications.

To put the porting effort in perspective, the Intel OmniPath Linux driver amounts to about 50K source lines of code (SLOC). From this codebase, the *PicoDriver* framework enabled us to port less than 3K SLOC to McKernel, which demands a significantly lower development effort than if we were to port the entire driver. We also emphasize that identically to the original IHK/McKernel, the *PicoDriver* architecture runs unmodified Linux binaries without any need for recompilation or specific communication libraries.

The rest of this paper is organized as follows. We begin with background information on multi-kernels and the OmniPath device driver 2. We describe the design and implementation of *PicoDriver* in Section 3. Evaluation is provided in Section 4. Section 5 discusses related work, and finally, Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

This section lays the groundwork for the proposed driver architecture by providing background information on the IHK/McKernel lightweight multi-kernel OS [13–15] and the organization of Intel's OmniPath network stack [5].

2.1 IHK/McKernel

An architectural overview of the main system components in IHK/-McKernel is depicted in Figure 1. A low-level software infrastructure, called Interface for Heterogeneous Kernels (IHK) [39], provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. IHK is capable of allocating and releasing host resources dynamically and no reboot of the host machine is required when altering configuration. The latest version of IHK is implemented as a collection of Linux kernel modules without any modifications to the Linux kernel itself. This enables relatively 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.

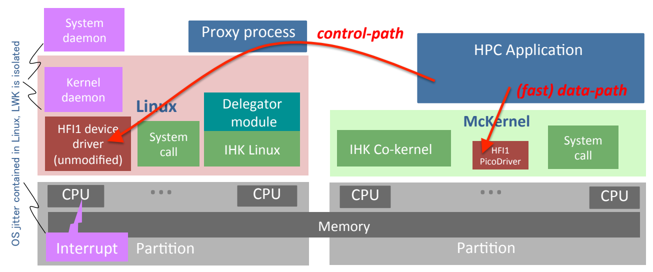


Figure 1: Overview of the IHK/McKernel architecture with the HFI1 *PicoDriver*.

McKernel is a lightweight co-kernel developed on top of IHK. It is designed explicitly for high-performance computing workloads, but 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 provides its own memory management, it supports processes and multi-threading, it has a simple round-robin co-operative (tick-less) scheduler, and it implements 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 exists a process in 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 [14]. 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. A detailed description of the device mapping mechanism is provided in [15].

Recently we have demonstrated that lightweight multi-kernels can indeed outperform Linux on various HPC mini-applications when evaluated on up to 2,000 Intel Xeon Phi nodes interconnected by Intel's OmniPath network [13]. At the same time, we also learnt that under certain circumstances the system call offloading model can introduce notable performance degradation on network operations. This is particularly true in case of the OmniPath network (which we will describe in Section 2.2), as it requires device driver interaction for a number of communication operations that are in the critical path of execution. In Section 4, we will provide detailed analysis on the impact of the offloading mechanism.

2.2 Omni-Path Fabric Architecture

The software support for Intel's OmniPath fabric is comprised of two main components in Linux based systems [5]. At the user level, the Performance Scaled Messaging (PSM) library provides low-level APIs for applications and for communication libraries, such as MPI implementations. The PSM library in turn interacts with the Intel Host Fabric Interface (HFI) driver in the Linux kernel via system calls invoked on the HFI device file. An architectural overview of the OmniPath driver stack is provided in Figure 2.

2.2.1 Intel Performance Scaled Messaging Library. The Performance Scaled Messaging library is a low-level user space communications interface to the Intel Omni-Path high-performance communication architecture. PSM offers an endpoint based communication model where an application can establish a connection to another node and drive data transfer operations on top of the matched queues (MQ) facility of the library. PSM provides two transfer modes both for send and receive operations. Send can be done via programmed I/O (PIO) or send DMA (SDMA). PIO optimizes latency and message rate for small messages and is entirely driven from user-space. SDMA optimizes bandwidth for large messages, utilizing 16 SDMA engines for CPU offload. SDMA operations require interaction with the device driver via the `writev()` system call on the device file. The default configured message size threshold above which SDMA is used is 64KB in the PSM library.

As for the receive operations, there is eager-receive and direct data placement. For eager-receive, data is received in library internal buffers first and copied to application buffers later. No handshake in advance is needed for eager-receive. Additionally, there is direct data placement to application buffers, where a handshake in advance is required. Under the hood, the library utilizes `ioctl()` system calls to register user-space buffers with the kernel driver. As one might expect, we will be focusing on SDMA send and the registration of user buffers for reception as these operations require system call involvement, which in turn trigger the offloading mechanism in the IHK/McKernel multi-kernel environment.

2.2.2 Intel OmniPath Host Fabric Interface Driver for Linux. The OmniPath device driver for Linux is called the Host Fabric Interface (HFI) driver. As device drivers generally do, the HFI driver exposes fabric functionalities through file operations and by memory mapping part of the PCI device into userspace. Specifically, the device driver provides an implementation for the following POSIX system calls: `open()`, `writev()`, `ioctl()`, `poll()`, `mmap()`, `lseek()` and `close()`.

We have already mentioned that we are primarily interested in the SDMA send operation, i.e., the implementation of the `writev()` call and the registration of expected receive, which is performed in `ioctl()`. However, it is worth pointing out that the driver implements a whole lot of other functionality besides these. This is important because - as we will see below - our intention is to directly reuse these via system call offloading, without any modifications to the Linux driver code. To better motivate the feasibility

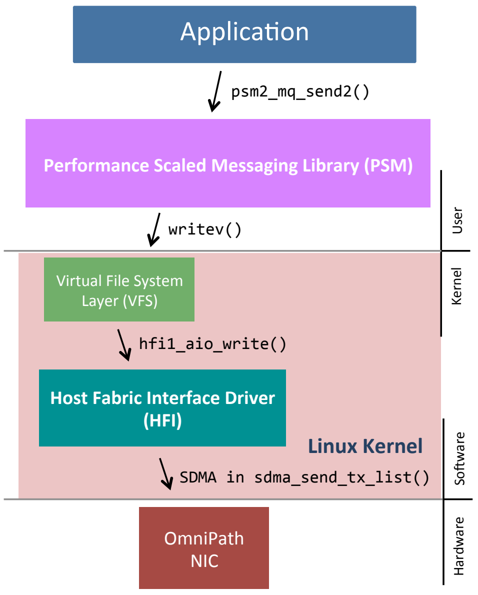

Application
psm2_mq_send2()
Performance Scaled Messaging Library (PSM)
writev()
User
Kernel
Virtual File System Layer (VFS)
hfi1_aio_write()
Host Fabric Interface Driver (HFI)
Linux Kernel
SDMA in sdma_send_tx_list()
Software
OmniPath NIC
Hardware

Figure 2: OmniPath software stack architecture in Linux.

of *PicoDriver*, we briefly describe the steps that are taken in the SDMA send and the reception buffer registration operations.

The `writev()` call for initiating an SDMA transfer passes an array of I/O vectors to the kernel. The first of these describes metadata about the operation and the rest provides information regarding the buffers that are about to be transferred. The driver internally verifies the buffers and calls `get_user_pages()` on the specified virtual ranges to obtain the physical pages that back the buffers. This also ensures that the pages are pinned in memory and cannot be swapped out in the meantime. The driver then reserves an SDMA engine and iterates the physical pages translating the physical addresses into SDMA transfer requests. These request structures are submitted to the SDMA engine's ring buffer along with the creation of metadata structures that represent the transfers. Completion notification for an SDMA transfer is performed in an interrupt handler that is executed when the corresponding IRQ is raised by the hardware. The IRQ handler uses callback functions to perform notification as well as cleanup of the associated metadata.

The registration of expected receive buffers is performed in `ioctl()`. By large, the execution steps of this operation are similar to that of the SDMA transfer, except that the physical addresses are translated into so called *RcvArray* entries which in turn are programmed to the hardware (i.e., written to specific offsets in the device mapping). User-space identifies the reception by TID identification numbers which can be also used to unprogramm the associated entries, i.e., to unregister user buffers. Note that the `ioctl()` call implements over a dozen different functionalities in the HFI driver, but only three from those are related to reception buffer registration.

In summary, both of the aforementioned operations translate user provided virtual address ranges to a hardware specific representation which in turn are submitted to the network interface card (NIC).

3 DESIGN AND IMPLEMENTATION

This section describes the design and implementation of *PicoDriver*. While we primarily focus on OmniPath, design decisions have been made with generality in mind. We emphasize that the changes introduced should be applicable for supporting other drivers as well, which in fact is part of our future plans.

For now, our primary goal is to create a framework that enables porting only a small piece of the HFI device driver (i.e., the performance sensitive routines discussed in the previous section) in a fashion that we can keep utilizing the original Linux driver for the rest of its functionality. Figure 1 shows the overall structure of *PicoDriver*. As seen, McKernel contains a small HFI *PicoDriver* while Linux hosts the original driver provided by Intel. McKernel, for instance, has no facilities such as the Linux device model, the VFS layer or `/proc` and `/sys` pseudo file systems. McKernel also doesn't provide mechanisms for bottom half processing (e.g., tasklets or workqueues [10]), often used in device drivers. However, it is not our intention to provide these mechanisms, but instead we strive to implement an LWK optimized version of the fast-path code. As we mentioned earlier, the HFI1 driver amounts to about 50K source lines of code (SLOC), but from this codebase, less than 3K SLOC is related to the routines that is in our interest, which demands a significantly lower development effort than if we were to port the entire driver. As shown, this architecture resembles OS bypass (i.e., bypassing Linux in this case) for the fast data-path of the driver, while leaving the control-path untouched in the unmodified Linux driver.

The most basic requirement for co-operating with the Linux device driver from McKernel is the ability to access Linux internal data structures. For this reason, before we further discuss the design of *PicoDriver*, we describe the changes to McKernel's virtual address space layout that were necessary to enable mutual access to kernel internals between Linux and the LWK.

3.1 Virtual Address Space Layout

McKernel is a completely independent kernel from Linux, i.e., it runs its own ELF image and it maintains its own set of virtual to physical mappings. The original McKernel virtual address space layout was designed with the plain objective of supporting a Linux compatible ABI. There are multiple, logically distinct ranges in the kernel virtual address space that typically appear in operating systems. The three main ones are the kernel image (i.e., the TEXT, BSS and DATA in case of ELF images), the direct mapping of physical memory and a dynamically managed range that is used for device mappings (called the `vmalloc()` range in Linux). Indeed, since the virtual address space is enormous in 64 bit addressing mode[1], it is common practice in OS kernels to map the entire physical memory upfront. Linux for example has a 64TB dedicated area just for this purpose on x86_64 platforms and this is where `kmalloc()` allocations are served from. Initially, we mapped the McKernel ELF image to the same address where the Linux image resides and had the dynamic range overlap with the one in Linux.

With *PicoDriver*, we faced the following challenges:

- TEXT, BSS and DATA segments of the two kernel images should not overlap.

[1]On current x86_64 hardware, the actually addressable space is only 48 bits.

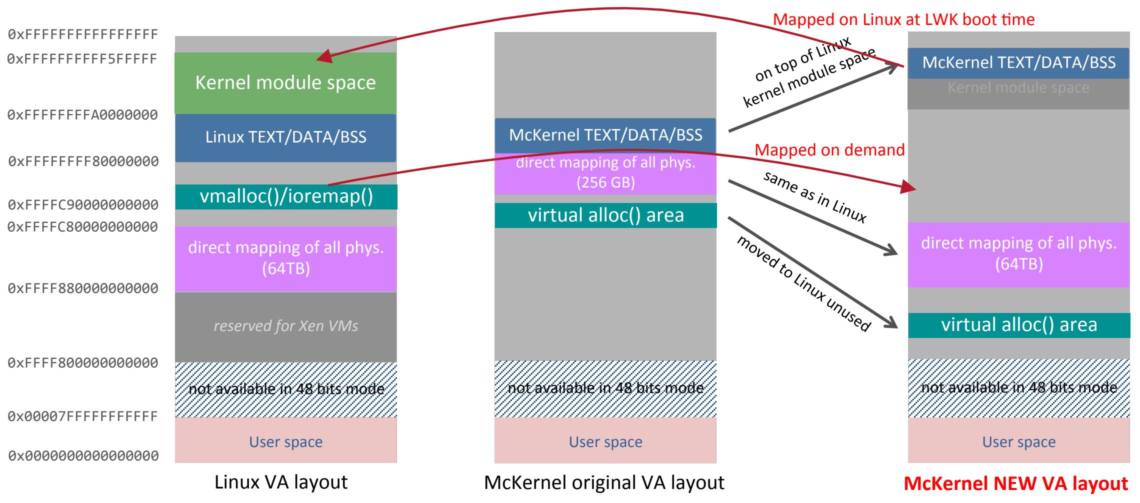


Figure 3: Virtual address space layout in Linux, the original McKernel and the one modified for *PicoDriver* (under x86_64).

- Dynamically allocated Linux data structures must be visible in McKernel (and vice versa) so that pointers can be dereferenced, i.e., they must follow the same virtual to physical mappings.
- Linux must be able to see McKernel's TEXT (so that it can access callback functions).

To address each of these requirements we introduced the following modifications. First, we moved the McKernel ELF image (in McKernel's address space) from its original location to the top of the Linux module space. This ensures that the TEXT/BSS/DATA segments of McKernel do not overlap with the ones in Linux. Second, we shifted the direct mapping of physical memory to the same address range in McKernel as in Linux, basically ensuring that any allocation from Linux' `kmalloc()` will be valid also in McKernel, and vica versa. Finally, we established a mapping of McKernel's ELF image also in the Linux kernel (initialized at the time of booting the LWK), which enables us to access McKernel functions in Linux. For address range reservation we rely on Linux' `vmap_area` that is used for managing module mappings. Figure 3 shows the virtual address space layout in Linux, in the original McKernel, as well as in the version prepared for *PicoDriver*. It also presents and overview of the modifications themselves. Note that although the virtual addresses displayed at the left side of the figure are accurate, the size of each range on the figure is not proportional to the actual virtual area and they simply serve the purpose of demonstration.

Overall, these changes provide a high degree of address space unification between the two kernels, making McKernel behave almost as a regular Linux kernel module. Nevertheless, it is important to see that none of these modifications degrade the LWK's full control over its assigned resources. McKernel assigned CPU cores are still invisible in Linux (i.e., offlined) and Linux has absolutely no control over McKernel's memory mappings or any other internal kernel mechanisms for that matter. To some extent, this address space unification resembles the proxy process' role in unifying user-space mappings, only that it is applied to the kernel virtual address space.

3.2 DWARF based Structure Extraction

Having gained access to Linux kernel memory mappings enabled us to dereference pointers to arbitrary Linux data structures including those that belong to the HFI driver. Note that the internal HFI data structures are initialized by Linux in offloaded system calls to the driver (i.e., at the time of calling `open()` on the device file) and are also used in other slow path calls (e.g., `mmap()` or `poll()`) that we do not intend to port to the LWK. Therefore, to be able to co-operate with original driver code we need to make sure that we access the correct fields at correct offsets. In summary, to interpret accesses correctly, code in McKernel needs to be aware of the structure layouts. It is worth mentioning that in most cases we only need a small subset of the fields in HFI data structures, since most of them are utilized by functionality that is exclusively executed in Linux and not in the LWK.

Our first approach was to manually copy the Linux headers and simply replace sub-structures we don't mimic such as Linux' `kobject` with a manually crafted character array. Essentially, porting data structures to the LWK manually. Although this approach works, it requires finding out how big each field is, which is not only laborious but also error prone since such items are likely to change across different versions of the driver. Some could even change depending on build options, which are not always obvious, and such errors would lead to runtime failures that are hard to diagnose.

Instead of trying to manually keep up to date with these changes, we opted for a different approach. Information about data structures, including the field positions, are in fact stored in the DWARF debugging information headers [11] of the module binary shipped by Intel. We inspect the binary and produce a header file containing only fields we are concerned with, automatically determining the type of each field as well as its location in the structure. We have developed a tool for this purpose, which we named *dwarf-extract-struct* [2]. The

[2]http://cgit.notk.org/asmadeus/dwarf-extract-struct.git/

tool systematically walks the DWARF headers until it finds the requested structure to extract (as `DW_TAG_structure_type`), and for each requested field it finds the appropriate `DW_TAG_member` from which we can obtain its offset (via `DW_AT_data_member_location`) and its type (through `DW_AT_type`). The DWARF header also conveniently stores the number of elements for arrays, or dereferences for pointer types.

```
struct sdma_state {
    union {
        char whole_struct[64];
        struct {
            char padding0[40];
            enum sdma_states current_state;
        };
        struct {
            char padding1[48];
            unsigned int go_s99_running;
        };
        struct {
            char padding2[52];
            enum sdma_states previous_state;
        };
    };
};
```

Listing 1: Automatically generated header for the HFI `sdma_state` structure.

The generated header is a C `struct` containing an unnamed `union`, with a character array of the size of the entire structure (so that the final size matches) and individual members each preceded with its independent padding. An example of such generated header for the HFI `sdma_state` structure is shown in Listing 1. Since the beginning of the development of *PicoDriver*, we have already updated twice to Intel's new releases. With the DWARF based header generation the porting effort has been on the order of hours.

3.3 Synchronization, Callbacks and Memory Management

Since interaction with the network device may happen simultaneously from Linux, e.g., through offloaded system calls and notification IRQs, and from McKernel via the HFI *PicoDriver*, correct synchronization between the two kernels is utmost important. Luckily, most of the synchronization points we had to deal with during the porting process utilizes `spin-locks`. As Linux and McKernel shares memory in a cache coherent fashion the only thing we had to ensure was that the spin lock implementation in the two kernels are compatible. This was not a major challenge as McKernel already adopted the Linux `spin-lock` implementation for the `x86_64` architecture. We also note that synchronization using the Linux `mutex` facility would be also feasible as internally the `mutex` implementation relies on `spin-locks`, however, we would likely simply spin in McKernel as opposed to sleep at the thread level because Linux would not be able to send wake up notifications across kernel boundaries. Finally, although we did not need it in this study, we have not solved the problem of RCU locks, which we left for future work.

As we mentioned above, the HFI driver gets notified of SDMA transfer completions by hardware interrupts. Since device interrupts are not handled by McKernel, Linux CPUs process all notifications. The driver code utilizes callback functions that are associated with each chunk of a transfer during SDMA request creation. Because data structures used in McKernel initiated requests are dynamically allocated in the LWK, we had to duplicate the callback and replace the deallocation routine with the one from McKernel. This new callback function exists in McKernel TEXT and thus Linux needs to be able to access function pointers in McKernel's ELF image. Section 3.1 explained how this mapping is implemented.

McKernel uses per-core data structures to provide a scalable memory allocator. When `kfree()` is called, the deallocated buffer is inserted to the per-core free memory list. Consequently, the memory allocator needs to be aware of the CPU core that is making the call. Because Linux CPUs are not managed by the LWK, an McKernel `kfree()` called on a Linux CPU would by default fail. We extended the memory management code of McKernel so that it recognizes when a deallocation routine is called on a Linux CPU and takes appropriate steps to handle it correctly. With the above modifications, SDMA completions can be safely processed on Linux CPUs. It is also worth pointing out that our synchronization methods retain the ability to share a single NIC across multiple LWKs and/or applications.

3.4 Optimization Opportunities

What makes the *PicoDriver* architecture rather powerful is that it provides opportunities for optimizing fast-path device driver operations to LWK internals. As we mentioned above, the original Linux HFI driver obtains information about user buffers using the `get_user_pages()` Linux kernel function, which returns the corresponding page structures that back the user supplied virtual range. The SDMA request submission routine in turn iterates these pages and translates the physical addresses to SDMA requests. The HFI network device accepts SDMA requests up to 10kB in size, assuming that the given physical memory range is contiguous. However, because page boundaries must to be checked carefully, at the time of writing this paper, the Linux HFI driver utilizes only up to `PAGE_SIZE` (i.e., 4kB on `x86_64`) long SDMA requests. This implies that not only is the driver unaware of contiguous physical memory ranges that cross page boundaries, but it even fails to recognize large page based mappings.

The principal policy of McKernel's memory management is to support `ANONYMOUS` memory mappings with contiguous (as much as possible) physical memory utilizing large page based translations. This implies that in the common case SDMA requests can be 10kB in size. Indeed, we have modified the HFI *PicoDriver* to efficiently handle large pages as well as contiguous physical memory that crosses page boundaries. Additionally, McKernel ensures that all `ANONYMOUS` mappings are pinned, i.e., they can be unmapped exclusively by user requested operations. This further simplifies SDMA transfer initialization because we merely need to iterate page tables instead of collecting references to page structures. In Section 4 we demonstrate how these modifications benefit data transfer performance.

4 EVALUATION

4.1 Experimental Environment

All application level evaluation were performed on Oakforest-PACS (OFP), a Fujitsu built, 25 peta-flops supercomputer installed at JCAHPC, managed by The University of Tsukuba and The University of Tokyo [20]. 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 our 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. From the operating system's perspective there are 272 logical CPUs organized around eight NUMA domains. Additionally, micro-benchmark level results were obtained using our dedicated development compute nodes, where we could instrument the HFI network driver for profiling information. These nodes are almost identical to OFP's compute nodes except that they are equipped with the 64 cores version of Xeon Phi™ CPU 7210.

The software environment was as follows. Compute nodes run XPPSL `1.5.1` with Linux kernel version `3.10.0-327.36.3`. XPPSL is a CentOS based distribution with a number of Intel supplied kernel level improvements specifically targeting the KNL processor. We used Intel MPI Version 2018 Update 1 Build 20171011 (id: 17941) in this study.

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. Our experience so far indicates that many applications need a power of two number of CPUs, or do not run faster on 66 or 68 cores.

We emphasize that for the Linux measurements we used 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 HFI measurements we deployed IHK and McKernel, commit hash `3bd05` and `da77a`, respectively. We utilized IHK's resource partitioning feature to reserve CPU cores and physical memory dynamically.

4.2 Benchmarks and Mini-applications

For the communication performance measurements we used Intel MPI's `IMB-MPI1` micro-benchmark [19]. As for application level results, we used the following mini-applications from the CORAL benchmark suite [9]. We also provide information regarding their runtime configuration.

- **LAMMPS** is a classical molecular dynamics code, and an acronym for Large-scale Atomic/Molecular Massively Parallel Simulator [34]. We used weak-scaling for LAMMPS and ran 64 MPI ranks per node, where each rank contained two OpenMP threads.
- **Nekbone** is a thermal hydraulic proxy app that is based on the open source spectral element code, Nek5000, which is designed for large eddy simulation (LES) and direct numerical simulation (DNS) of turbulence in complex domains [25]. We used weak-scaling for Nekbone and ran 32 MPI ranks per node, where each rank contained four OpenMP threads.
- **UMT2013** is an LLNL ASC proxy application that performs three-dimensional, non-linear, radiation transport calculations using deterministic (Sn) methods [24]. We used weak-scaling for UMT2013 and ran 32 MPI ranks per node, where each rank contained four OpenMP threads.
- **HACC** is the Hardware Accelerated Cosmology Code framework that uses N-body techniques to simulate the formation of structure in collisionless fluids under the influence of gravity in an expanding universe [22]. We used weak-scaling for HACC and ran 32 MPI ranks per node, where each rank contained four OpenMP threads.
- **QBOX** is a first-principles molecular dynamics code used to compute the properties of materials directly from the underlying physics equations. Density Functional Theory is used to provide a quantum-mechanical description of chemically-active electrons and nonlocal norm-conserving pseudopotentials are used to represent ions and core electrons [23]. We used weak-scaling for QBOX and ran 32 MPI ranks per node, where each rank contained four OpenMP threads.

For applications that fit into MCDRAM entirely we ran them exclusively out of MCDRAM, and for those which did not (e.g., UMT2013), we prioritize MCDRAM, but fall back to DRAM when necessary. For MPI profiling we utilized Intel MPI's `I_MPI_STATS` environment variable.

4.3 Results

The first experiment we performed was a simple MPI ping-pong test to assess communication bandwidth in Linux, in McKernel and in McKernel with the HFI *PicoDriver*.

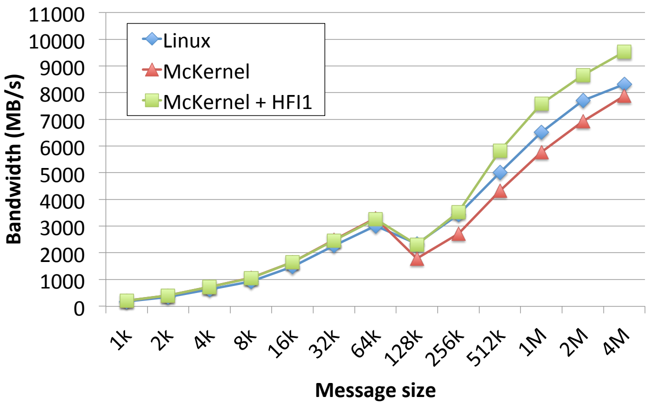
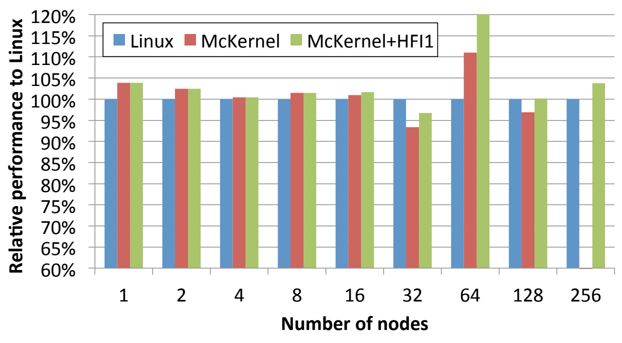
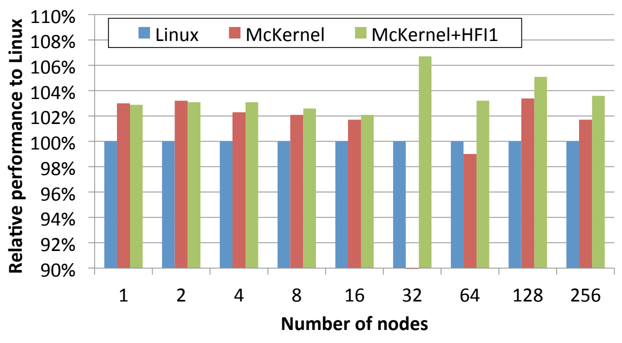


Figure 4: MPI Ping-pong bandwidth measurement.

Figure 4 shows the results of this experiment. As seen, the original McKernel version visibly suffers from syscall offloading overhead for large messages where data transfers involves system calls into the device driver, achieving only about 90% of the Linux performance. On the contrary, McKernel with the HFI *PicoDriver* outperforms Linux by up to 15% on 4MB buffer size. We have instrumented the HFI device driver and verified that the Linux driver submits only up to `PAGE_SIZE` (i.e., 4kB) long SDMA requests to the NIC. As we discussed before, McKernel attempts to back `ANONYMOUS` memory

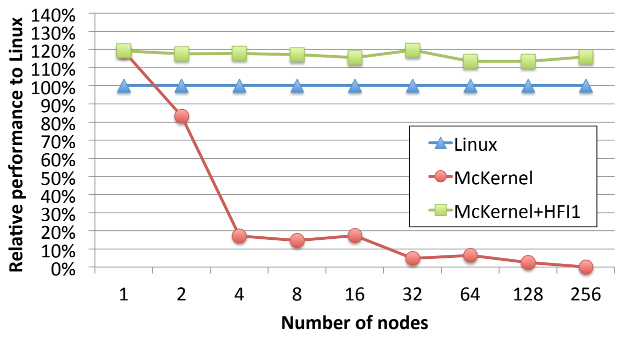


(a) LAMMPS

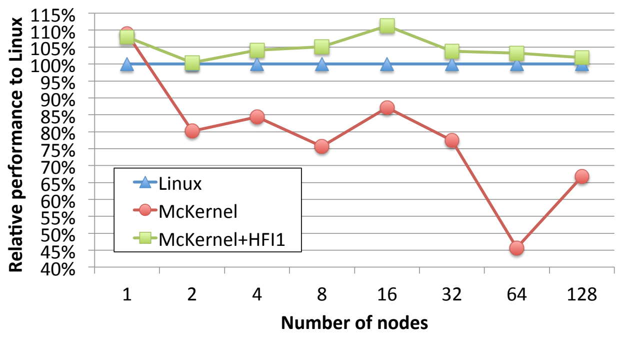


(b) Nekbone

Figure 5: Performance results for LAMMPS and Nekbone.



(a) UMT2013



(b) HACC

Figure 6: Performance results for UMT2013 and HACC.

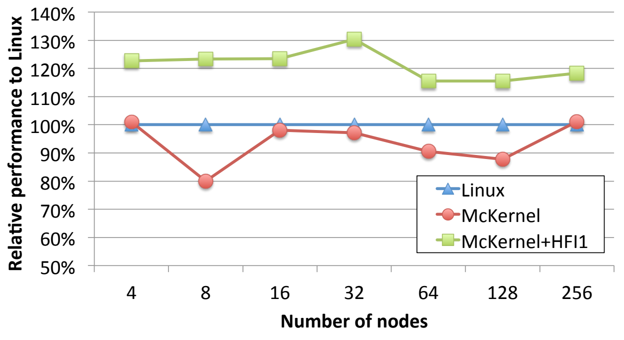


Figure 7: Performance results for QBOX.

mappings with physically contiguous memory whenever it is possible which significantly increases the probability of being able to use bigger SDMA requests. We have also verified that McKernel with the HFI driver consistently utilizes the maximum SDMA request size (i.e., 10kB per request) when the physical memory behind the mapping is contiguous. We believe that this is the main reason that constitutes to the observed performance improvement.

Let us turn our attention now to application level evaluation. We will be reporting numbers for Linux, the original McKernel and McKernel with the HFI *PicoDriver*. Note that the original McKernel and the HFI enabled version are two separate branches of development that had slightly diverged at the time of running our measurements. As we will see below, this had an impact on to the degree to which we could collect results. We also note that all application results are the average of multiple (at least three) runs, unless stated otherwise. Applications report figure of merit on a per-application basis. For clarity, instead of reporting absolute numbers we indicate relative performance to Linux. Linux values are included as well, displayed at 100%.

Before evaluating applications for which we did anticipate improvements, we also wanted to verify that the inclusion of the device driver does not impact applications adversely for which performance was already satisfactory. Therefore, we ran a number of tests to confirm this.

Figure 5 shows results for LAMMPS and Nekbone. LAMMPS performed similarly to Linux when running on the original McKernel, while for Nekbone we observed a small improvement for McKernel from the beginning on. Note that we do not have results for LAMMPS and Nekbone on the original McKernel version when running on

Table 1: Communication profile of UMT2013, HACC and QBOX on 8 compute nodes.

OS/	**Linux**				**McKernel**				**McKernel + HFI**			
App.	Call (MPI_)	Time	% MPI	% Rt	Call (MPI_)	Time	% MPI	% Rt	Call (MPI_)	Time	% MPI	% Rt
UMT2013	Barrier	1114.17	58.73	11.28	**Wait**	**17731.30**	49.29	40.35	Barrier	920.26	43.97	9.16
	Allreduce	297.14	15.66	3.01	Barrier	9223.53	25.64	20.99	Init	766.33	36.61	7.62
	Wait	**235.54**	12.42	2.38	Start	3944.92	10.97	8.98	**Wait**	**186.83**	8.93	1.86
	Init	140.65	7.41	1.42	Waitall	3856.68	10.72	8.78	Allreduce	160.46	7.67	1.60
	Request_free	47.26	2.49	0.48	Init	581.17	1.62	1.32	Waitall	42.98	2.05	0.43
HACC	Cart_create	15845.9	54.42	13.06	**Wait**	**40408.21**	67.73	26.45	Cart_create	4790.85	50.89	4.66
	Wait	**9905.76**	34.02	8.16	Waitall	6789.21	11.38	4.44	**Wait**	**2372.18**	25.20	2.31
	Allreduce	1414.18	4.86	1.17	Cart_create	5742.05	9.62	3.76	Init_thread	867.438	9.21	0.84
	Waitall	997.67	3.43	0.82	Recv	4757.27	7.97	3.11	Waitall	750.571	7.97	0.73
	Barrier	482.83	1.66	0.40	Init_thread	576.65	0.97	0.38	Allreduce	343.13	3.64	0.33
QBOX	Bcast	479.2	29.4	20.7	Bcast	1436.7	42.3	35.1	*Init*	*965.2*	47.8	35.1
	Init	*324.8*	19.9	14.1	*Init*	*568.9*	16.7	13.9	Bcast	268.9	13.3	9.7
	Alltoallv	279.8	17.1	12.1	Recv	533.4	15.7	13.0	Alltoallv	222.5	11.0	8.1
	Allreduce	159.1	9.7	6.9	Alltoallv	221.1	6.5	5.4	Allreduce	121.2	6.0	4.4
	Recv	139.1	8.5	6.0	Scan	145.1	4.2	3.5	Comm_create	95.6	4.7	3.4

256 and 32 nodes, respectively. This was due to an unresolved bug that has been fixed in the HFI driver branch but not yet backported to the master. Nevertheless, with the HFI driver version we could obtain results using nodes all the way up to 256. As the figure shows, McKernel with the HFI *PicoDriver* performs similarly to the original version, often slightly outperforming it on both of these workloads, which confirms our hypothesis that the new driver architecture will not introduce performance degradation to workloads that otherwise are not affected by the extra overhead of system call offloading.

On the other hand, there are a number of applications that truly motivated this work. UMT2013, HACC and QBOX have been heavily affected by the overhead of the HFI device driver involvement in the original McKernel architecture. It is worth pointing out that because we run up to 64 ranks per node, simultaneous interaction with the device driver via system call offloading is not only affected by the cost of offloading itself, but also by the fact that there are substantially lower number of Linux CPUs than the number of MPI ranks (i.e., four Linux CPUs vs. 32 or 64 MPI ranks). This further amplifies the cost of these calls because it introduces high contention on a few Linux CPUs for driver processing. At the same time, it also motivates the need to move fast-path driver code into the LWK.

Figure 6 and Figure 7 demonstrate the performance results we obtained for these applications. As seen, both UMT2013 and HACC performed on par with Linux when using a single compute node. However, both applications perform significantly lower than Linux when running multi-node executions. In fact, UMT2013 achieves less than 20% of the Linux performance when run on more than 4 compute nodes. Although not to the same degree, HACC also is slower, attaining only 71% in average of the Linux performance. On the other hand, with the HFI *PicoDriver* enabled in McKernel, we see substantial improvements across all configurations. McKernel with HFI consistently outperforms Linux on these workloads by up to 20%.

As for QBOX, due to the input configurations we had available, we were only able to run the application on at least 4 compute nodes. Notice that the `X` axis of Figure 7 starts from 4. Interestingly, QBOX did not perform significantly lower even on the original McKernel than Linux for all node counts. We have not investigated the route cause of this phenomena, nevertheless, McKernel with HFI provides substantial speedups achieving up to 30% improvement over Linux.

To gain a better understanding of what is behind the improvements, we took a communication profile on 8 nodes for Linux, McKernel and McKernel with HFI for these three applications. Note that due to the limited availability of dedicated compute nodes in SNC-4 mode on OFP, the profiling information we provide were obtained in Quadrant mode. Although SNC-4 mode performed slightly better than Quadrant in all cases, we observed almost identical relative performance among the different OS configurations in Quadrant mode as in SNC-4.

Table 1 summarizes our findings. It shows the top five most dominant MPI calls for each benchmark on all OS configurations. The $Time$ column reports cumulative time spent in the call summed over all ranks. The columns *%MPI* and *%Rt* report the ratio of the call from the time spent in MPI and from the overall runtime, respectively. There are a number of important observations in the table. For example, both for UMT2013 and for HACC, there is almost an order of magnitude more time spent in the top MPI calls compared between the original and the HFI enabled McKernel versions. To emphasize this observation, we have highlighted (as bold) the time spend in `MPI_Wait()` in the table as this routine is where communication progression for asynchronous data transfer is typically made. On the other hand, when compared to Linux, McKernel with the HFI *PicoDriver* spends visible less time in this operation (both in terms of absolute time as well as relative to the overall runtime), suggesting higher performance data transmission and more balanced execution among MPI ranks.

Another interesting observation is the time spent in `MPI_Init()`, which we highlighted (in italic) for QBOX. As seen, McKernel with

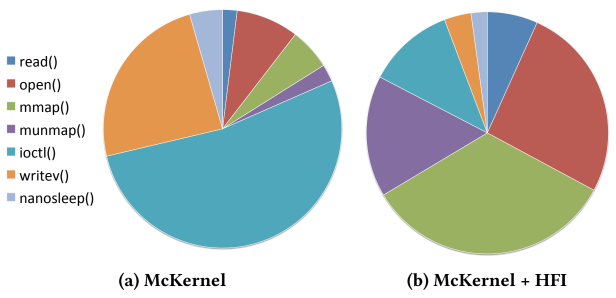


Figure 8: System call breakdown for UMT2013.

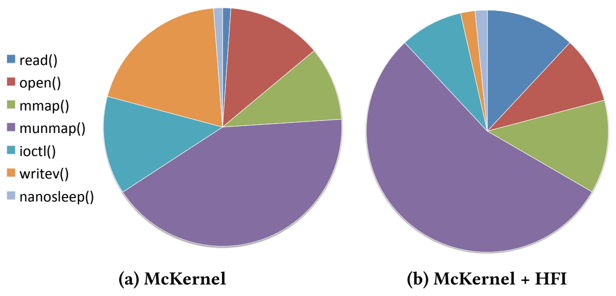


Figure 9: System call breakdown for QBOX.

HFI spends substantially more time in this call than the other OS configurations. Note that by default device initialization in `MPI_Init()` is offloaded in McKernel, which explains why it takes more time than in Linux. However, in McKernel with HFI an additional cost comes from the McKernel side initialization of kernel level mappings of device driver internals, etc. At the same time, McKernel with the HFI driver spends visibly less time in actual communication routines (e.g., `MPI_Bcast()` or `MPI_Alltoallv()`) compared to Linux. This disparity between the performance of critical and non-critical calls reflects well on the general idea of *PicoDriver*, i.e., to optimize for fast-path operations at the cost of other infrequent administrative invocations.

To further validate our claims, we also profiled McKernel with and without the HFI *PicoDriver* at the kernel level, for which we used our own in-house kernel profiler. This is currently only available for McKernel and unfortunately we can not provide identical measurements for Linux. Instead, we present a detailed breakdown of the time spent in system calls for the statistically most significant invocations compared between the two McKernel configurations.

For brevity, we provide measurements only for UMT2013 and QBOX. Figures 8 and 9 show the results. For each benchmark, the corresponding pie chart presents the times proportionally spent in the top seven system calls. To put these charts in context, the amount of time spent in kernel space when running McKernel with HFI is 7% and 25% of the original McKernel system time for UMT2013 and QBOX, respectively. The charts list the name of the calls at the left side of the figure. For UMT2013 (shown in Figure 8a) `ioctl()` and `writev()` dominates the time spent in the original McKernel, which reflects the offloaded SDMA send and the expected receive registration operations. Indeed, these two calls constitute over 70% of the time spent in kernel space. To the contrary, when the HFI *PicoDriver* is enabled, the relative cost of these calls is reduced below 30% in average. For QBOX, shown in Figure 9, we observe similar tendency with respect to the `ioctl()` and `writev()` calls, however, the kernel level profiler also revealed that `munmap()` dominates the system level cost, which leaves us with the opportunity to further optimize memory management in the future.

5 RELATED WORK

Without striving for completeness, this section discusses related studies in the domain of operating systems for HPC, as well as proposals related to device driver architectures.

5.1 Operating Systems in HPC

Lightweight kernels (LWKs) [37] tailored for HPC workloads date back to the early 1990s. These kernels ensure low operating system noise, excellent scalability and predictable application performance for large scale HPC simulations. One of the first LWKs that has been deployed in a production environment was Catamount [21], developed at Sandia National laboratories. IBM's BlueGene line of supercomputers have also been running an HPC specific LWK called the Compute Node Kernel (CNK) [17]. While Catamount has been written entirely from scratch, CNK borrows a substantial amount of code from Linux so that it can better support standard UNIX features. The most recent of Sandia National Laboratories' LWKs is Kitten [32], which also distinguishes itself from their prior LWKs by providing 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 [29, 35] and ZeptoOS [43] follow this path. The general approach here is to eliminate daemon processes, simplify the scheduler, and replace the memory management system. Linux' complex code base, however, can make it difficult to evict all undesired effects. In addition, it is also cumbersome to maintain Linux modifications with the rapidly evolving Linux source code.

Recently, with the advent of many-core CPUs, a new multi-kernel based approach has been proposed [15, 26, 30, 42]. The basic idea of multi-kernels is to run Linux and an LWK side-by-side on different cores of the CPU and to provide OS services in collaboration between the two kernels. FusedOS [31] was the first proposal to combine Linux with an LWK. It's primary motivation was to address CPU core heterogeneity between system and application cores. Contrary to McKernel, FusedOS runs the LWK at user level. In the FusedOS prototype, the kernel code on the application core is simply a stub that offloads all system calls to a corresponding user-level proxy process called CL. The proxy process itself is similar to that in IHK/McKernel, but in FusedOS the entire LWK is implemented within the CL process. The FusedOS work was the first to demonstrate that Linux noise can be isolated to the Linux cores and avoid interference with the HPC application running on the LWK cores. This property has been also one of the main driver for the McKernel model.

From more recent multi-kernel projects, one of the most similar efforts to ours is Intel's mOS [16, 42]. The most notable difference between McKernel and mOS is the way how LWK and Linux are integrated. mOS takes a path of much stronger integration with the motivation of easing LWK development and to directly take advantage of the Linux kernel infrastructure. Nevertheless, this approach comes at the cost of Linux modifications and an increased complexity of eliminating OS interference. On the other hand, it allows potentially calling into Linux device drivers on LWK cores.

Hobbes [6] was another of the DOE's Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to better support application composition. Hobbes also utilizes virtualization technologies to provide the flexibility to support requirements of application components for different node-level OSes. At the bottom of the software stack, Hobbes relies on Kitten [32] as its LWK component, on top of which Palacios [27] serves as a virtual machine monitor. As opposed to IHK/McKernel, Hobbes separates Linux and Kitten at the PCI device level, which poses difficulties both for supporting full POSIX API and the necessary device drivers in the LWK.

Argo [4] is an exa-scale OS project targeted at applications with complex work-flows. The Argo vision is using OS and runtime specialization (through enhanced Linux containers) on compute nodes. Argo expects to use a ServiceOS like Linux to boot the node and run management services. It then runs different container instances that cater to the specific needs of applications.

5.2 Novel Device Driver Architectures

Device driver architectures for operating systems have been studied extensively in the literature. One of the proposed architectures that resembles *PicoDriver* is Nooks [40], although Nooks' primary motivation is to provide an architecture for reliability. Device drivers in Nooks execute in the kernel address space, but within different protection domains. Thus, a device driver may dereference pointers supplied by the kernel without copying the data or translating addresses, but virtual memory protection and lowered privilege levels are used for isolating and recovering faulty code. Similarly to Nooks, *PicoDriver* also runs the fast-path code in another virtual memory protection domain, i.e., in McKernel's kernel space.

Microdrivers [12] is another driver proposal that has a split architecture similar to *PicoDriver*. Microdrivers split driver functionality between a small kernel-mode component and a larger user-mode component, which reduce the amount of driver code running in the kernel and thus, reduces the likelihood that a faulty driver code would corrupt other sensitive kernel data. Again, the motivation of Microdrivers is reliability and thus it is different from *PicoDriver*, however, Microdrivers' *code generator* module deals with similar issues to that of *PicoDriver*'s DWARF structure extractor.

Another interesting device driver architecture explicitly targeting multi-core environments was proposed in [18], where the basic idea is to split the driver stack into multiple components and run them on different CPU cores of a multi-core system. This philosophy resembles *PicoDriver*'s architecture as it also distinguishes CPU cores for slow-path and fast-path components of the device driver.

With respect to device drivers in recent lightweight kernels, Lange et. al. discussed how the Kitten operating system provides various kernel level interfaces that mimic the Linux device driver infrastructure [27]. Although these interfaces help for porting entire device drivers to LWKs, they require significantly more development effort than porting only the fast-path piece of a driver. The same study also demonstrates how virtualization may be utilized to exploiting existing device drivers by using passthrough I/O to virtual machines.

Finally, Arrakis proposed exposing data-plane operations directly to applications relying on hardware virtualization features of the NIC [33]. Their approach provides a similar split architecture to *PicoDriver*, although their target environment is not high-performance computing.

6 CONCLUSION AND FUTURE WORK

This paper has presented *PicoDriver*, a novel device driver architecture for lightweight multi-kernel operating systems in high-performance computing environments. *PicoDriver* eases device driver development for LWKs by allowing fast-path code to be ported to an LWK in a straightforward fashion. *PicoDriver* requires no modification to the original Linux driver, yet, it enables optimization opportunities in the LWK. We have discussed the design of *PicoDriver* and the challenges we faced during the implementation of Intel OmniPath network driver in the IHK/McKernel multi-kernel operating system. In essence, *PicoDriver* extends the lightweight multi-kernel philosophy of optimizing fast-path operations at the expense of infrequent administrative calls to the device driver domain. On a number of mini-applications we have demonstrated that McKernel with *PicoDriver* can outperform Linux by up to 30% when deployed on up to 256 Intel Xeon Phi KNL compute nodes, interconnected by Intel's OmniPath network.

Our immediate future work is to address the memory management shortcomings of McKernel that have been revealed by profiling during the experiments of this study. In the near future, we have also plans to perform a much larger scale evaluation of McKernel using the *PicoDriver* framework. Finally, we intend to further extend this work by porting memory registration routines from the Mellanox Infiniband driver.

ACKNOWLEDGMENT

This work has been partially funded by MEXT's program for the Development and Improvement of Next Generation Ultra High-Speed Computer Systems under its subsidies for operating the Specific Advanced Large Research Facilities in Japan.

We acknowledge Prof. Taisuke Boku from The University of Tsukuba and Prof. Nakajima Kengo from The University of Tokyo for their support to let us access the OFP supercomputer. We also thank the Intel mOS team for providing us with valuable information on the configuration of the mini-applications we used for evaluation. Finally, we acknowledge Takahiro Ogura and Toshiro Saiki from Fujitsu for their relentless support regarding the operation of the OFP machine.

REFERENCES

[1] R. Alverson, D. Roweth, and L. Kaplan. 2010. The Gemini System Interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*. 83–87. https://doi.org/10.1109/HOTI.2010.23

[2] InfiniBand Trade Association. 2016. InfiniBand Architecture Specification, Release 1.3.1. (Nov 2016).

[3] BDEC Committee. 2017. The BDEC "Pathways to Convergence" Report. http://www.exascale.org/bdec/. (15 Nov. 2017).

[4] 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. (March 2017).

[5] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. 2015. Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI '15)*. IEEE Computer Society, Washington, DC, USA, 1–9.

[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 (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 (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] CORAL. 2013. Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/. (Nov. 2013).

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc.

[11] DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format Version 5. http://dwarfstd.org/. (2017).

[12] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. 2008. The Design and Implementation of Microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 168–178. https://doi.org/10.1145/1346281.1346303

[13] Balazs Gerofi, Rolf Riesen, Masamichi Takagi, Taisuke Boku, Yutaka Ishikawa, and Robert W. Wisniewski. 2018 (to appear). Performance and Scalability of Lightweight Multi-Kernel based Operating Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[14] 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)*.

[15] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. 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)*. 1041–1050.

[16] 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*. ACM, Article 5. https://doi.org/10.1145/2768405.2768410

[17] 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)*.

[18] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. 2012. Keep net working - on a dependable and fast networking stack. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. https://doi.org/10.1109/DSN.2012.6263933

[19] Intel Corporation. 2018. Intel MPI Benchmarks. https://software.intel.com/en-us/articles/intel-mpi-benchmarks. (Feb. 2018).

[20] Joint Center for Advanced HPC (JCAHPC). 2017. Basic Specification of Oakforest-PACS. http://jcahpc.jp/files/OFP-basic.pdf. (March 2017).

[21] Suzanne M. Kelly and Ron Brightwell. 2005. Software architecture of the light weight kernel, Catamount. In *Cray User Group*. 16–19.

[22] Lawrence Livermore National Lab. 2017. HACC Summary v1.5. https://asc.llnl.gov/CORAL-benchmarks/Summaries/HACC_Summary_v1.5.pdf. (27 Jan. 2017).

[23] Lawrence Livermore National Lab. 2017. Qbox (qb@llbranch). ://asc.llnl.gov/CORAL-benchmarks/Summaries/QBox_Summary_v1.2.pdf. (27 Jan. 2017).

[24] Lawrence Livermore National Lab. 2017. UMT Summary Version 1.2. https://asc.llnl.gov/CORAL-benchmarks/Summaries/UMT2013_Summary_v1.2.pdf. (27 Jan. 2017).

[25] Argonne National Laboratory. 2017. Proxy-Apps for Thermal Hydraulics. https://cesar.mcs.anl.gov/content/software/thermal_hydraulics. (27 Jan. 2017).

[26] 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 (ROSS '16)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2931088.2931095

[27] 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

[28] 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.

[29] 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*.

[30] 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 (HPDC '15)*. ACM, New York, NY, USA, 149–160.

[31] 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

[32] 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.

[33] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4, Article 11 (Nov. 2015), 30 pages. https://doi.org/10.1145/2812806

[34] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-range Molecular Dynamics. (March 1995), 19 pages. https://doi.org/10.1006/jcph.1995.1039

[35] 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)*.

[36] 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), 793–817. https://doi.org/10.1002/cpe.v21:6

[37] 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 (ROSS)*. ACM, New York, NY, USA. https://doi.org/10.1145/2768405.2768414

[38] 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 (Supercomputing '94)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 580–589.

[39] 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)*.

[40] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. 2002. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop (EW 10)*. ACM, New York, NY, USA, 102–107.

[41] 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.

[42] 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 (ROSS)*. ACM, New York, NY, USA, Article 2.

[43] Kazutomo Yoshii, Kamil Iskra, Harish Naik, Pete Beckmann, 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.