Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures

Taku Shimosawa*, Balazs Gerofi†, Masamichi Takagi‡, Gou Nakamura§, Tomoki Shirasawa¶, Yuji Saeki*, Masaaki Shimizu*, Atsushi Hori‡ and Yutaka Ishikawa†‡

*Hitachi, Ltd.

†Graduate School of Information Science and Technology, The University of Tokyo

‡RIKEN Advanced Institute for Computational Science

§Hitachi Solutions, Ltd.

¶Hitachi Solutions East Japan, Ltd.

***Abstract*—Turning towards exascale systems and beyond, it has been widely argued that the currently available systems software is not going to be feasible due to various requirements such as the ability to deal with heterogeneous architectures, the need for systems level optimization targeting specific applications, elimination of OS noise, and at the same time, compatibility with legacy applications. To cope with these issues, a hybrid design of operating systems where light-weight specialized kernels can cooperate with a traditional OS kernel seems adequate, and a number of recent research projects are now heading into this direction.**

This paper presents *Interface for Heterogeneous Kernels (IHK)*, a general framework enabling hybrid kernel designs in systems equipped with manycore processors and/or accelerators. IHK provides a range of capabilities, such as resource partitioning, management of heterogeneous OS kernels, as well as a low-level communication layer among the kernels. We describe IHK's interface and demonstrate its feasibility for hybrid kernel designs through executing various different light-weight OS kernels on top of it, which are specialized for certain types of applications. We use the Intel Xeon Phi, Intel's latest manycore coprocessor, as our experimental platform.

***Keywords*-OS kernel; heterogeneous kernels; manycore processors, OS abstraction; inter-kernel communication**

I. Introduction

As the rate of CPU clock improvement has stalled for the last decade primarily due to energy consumption issues, increased use of parallelism in the form of multi- and many-core processors have been chased to improve overall performance. Manycore processors, which come with a large number of CPU cores providing relatively lower clock rates or limited functionality, but significantly higher power efficiency are already widespread in high performance computing. A typical example of such processors is the Xeon Phi [2], Intel's latest design targeting parallel workloads. The Xeon Phi provides over 60 of low-frequency x86 cores which are all capable of running operating system (OS) code. Two systems in the top ten supercomputers as of November 2013 [3] are already based on the Xeon Phi. The current Xeon Phi comes in the form of a PCI Express attached coprocessor and requires a host machine, but the next generation Xeon Phi chip will be available in a standalone format as per Intel's latest announcement [4]. Moreover, projections for future exascale configurations suggest various types of manycore architectures, foreseeing hardware features such as CPU cores with different frequencies and/or different core architectures, CPU cores with multiple domains of memory coherency, and cores with multiple memory chips on independent buses.

Traditional operating systems, such as Linux, have proved to have mainly four issues when they are to be used for high performance computing in manycore systems. First of all, OS services can affect the performance of applications through cache pollution and the introduced fluctuation in scheduling. This effect is called "OS jitter" or "OS noise" [5], and known to be a major contributor of performance degradation especially in large scale parallel systems. Even if applications and OS are bound to different cores, OS services should be available for the applications, thus there remains needs for special mechanisms for OS services that alleviates OS noises. Second, a single operating system kernel running on all CPU cores in a heterogeneous system can result in degraded performance on the low performance cores. In this scenario, the operating system needs the ability to forward OS requests from the low performance cores to be handled by the kernel running on a high performance core. Third, traditional operating systems assume cache-coherent shared memory across all CPU cores to store their code and data. In a system with multiple coherency domains, this may either come with large overhead or may not work properly due to the fact that data must be transferred explicitly between coherence domains. Fourth, as traditional operating systems are made to be generic in the sense that they are designed to perform moderately well for a vast types of computers, it is very likely that difficulties will arise when functionalities targeting certain types of HPC applications and/or certain types of hardware are introduced. The number of lines in the Linux source code is over 16 million at

[0]This work is based on the Ph.D dissertation[1] of the first author.

the time of writing this paper, and it already requires a tremendous effort to apply fundamental changes, such as replacing parts of the virtual memory subsystem.

The development of specialized new runtimes and OS kernels is motivated not only by the above mentioned issues, but also by the demand for tighter orchestration of components across the entire software stack (i.e., application, runtime and OSes), resiliency, fault-tolerance, and fine-grained power management [6]. In addition, the long-established OS interfaces should be also supported so that legacy applications can still benefit from newer hardware.

A large number of light-weight kernels for HPC applications, such as Catamount [7] and CNK [8], have been developed and used along with full-weight kernels running in separate nodes. Targeting exascale systems, several recent research efforts including FusedOS [9], Argo [6] and mOS [10] seek ways to leverage new hardware capacity by providing and running multiple heterogeneous kernels in a single node. Running two types of kernels at the same time, a full-weight kernel such as Linux to provide the established interfaces for applications, and a light-weight kernel specialized for HPC applications, such as the kernels described above, is a promising way to build exascale systems.

Although there is an increasing demand for specialized light-weight kernels, to the best of our knowledge, existing software infrastructure still lacks a general framework for supporting the development and the execution of heterogeneous light-weight OS kernels. Such framework would not only allow to easily boot different application specific light-weight kernels on the same machine, but it could also enable rapid development of new OS designs by providing the basic facilities for CPU initialization and inter-kernel communication, which often is a significant part of the development effort for a multikernel configuration.

In this paper, we propose Interface for Heterogeneous Kernels (IHK), a minimalistic, but general framework that allows management of heterogeneous OS kernels running over separate CPU cores of a manycore CPU or accelerator(s). IHK's main responsibilities are resource partitioning, i.e., assigning certain CPU cores and ranges of physical memory to specific kernels; management of heterogeneous OS kernels, such as loading kernel images and initiating boot/shutdown; and a low-level inter-kernel communication mechanism that enables kernels to communicate with each other even if they reside in different cache coherency domains. Nevertheless, the target of IHK is not limited to communication among CPU cores on the same node, but also to communication among cores across nodes.

This paper presents the design of IHK, and the implementations of three different types of light-weight kernels built on top of it: McKernel, a Hybrid Segmentation Kernel [11] and one for addressing hierarchical memory management [12]. McKernel aims at providing a noiseless environment for applications with backward compatibility by delegating system call requests to a master Linux kernel. The latter two kernels are examples of special purpose light-weight kernels for satisfying particular application needs.

This paper makes the following contributions:

- Designing a general framework to partition resources, manage, and execute multiple heterogeneous kernels in a single node.
- Presenting a communication facility between the heterogeneous kernels running in a manycore processor.
- Demonstrating the generality of IHK through three types of light-weight kernels.

The rest of this paper is structured as follows: the next section describes our target systems, and discusses existing hybrid operating systems for HPC applications. Requirements and design policies of a general framework for hybrid OS kernels are presented in Section III. Following the policy, the actual design of IHK is provided in Section IV. IHK implementation details and McKernel are introduced in V. A brief description of the two special purpose kernels running on top of IHK, are presented in Section VI. Section VII evaluates IHK and McKernel. Section VIII describes related work, and Section IX concludes the paper.

II. Background

A. Target Systems

We have several assumptions with regards to possible differences between future many-core systems and the currently available multi-cores. First, there may be multiple types of cores on a single die. For example, general-purpose cores optimized for single threaded execution could be accompanied with computation cores specialized for parallel workloads (such as GPGPUs). Alternatively, cores with different operating frequencies like the ARM big.LITTLE [13] might be available. Second, manycore systems might have multiple separated memory address spaces, where some of the memory spaces may not be accessible directly from a certain set of cores. Even if accessible, latency may vary depending on which core accesses which memory area and cache coherency may not be retained, such as in case over PCI-Express. Third, due to the large number of cores, parallel applications running within a single processor may be more sensitive to jitter since the effect of jitter increases with the number of cores. Finally, we also assume that cores can execute privileged code (i.e., OS code) unlike in current GPUs or other type of special purpose accelerators.

One of our current target platforms is the Intel Xeon Phi, which comes with 61 x86 CPU cores (4-way SMT each), providing 244 hardware threads altogether. Although these cores are homogeneous, the co-processor currently needs to be attached to a regular host CPU which has normally higher clock frequency. Furthermore, the Xeon Phi has its own local GDDR5 memory in addition to the main memory of the host. The Xeon Phi's local memory and the host DRAM are accessible from both the Xeon Phi and the host CPU, but accessing remote memory costs more due to the overhead of the PCI Express bus.

B. Hybrid Kernels Approach

Hybrid kernels have been proposed and implemented both in research and in production. For example, combination of I/O nodes and compute nodes were deployed in several supercomputer systems including IBM's Blue Gene series [14], where a specialized light-weight kernel (CNK) is running on the compute nodes and Linux is running on the I/O nodes. The same approach can be taken inside a manycore processor. FusedOS [9] was the first proposal to promote the integration of a commodity operating system with rich functionalities and a light-weight kernel that actually runs the applications. Although several recent efforts, such as mOS [10] or Argo [6], follow a similar path, none of these studies provide a general framework for running different types of light-weight kernels. We believe that the ability to support heterogeneous light-weight kernels is highly important because it allows rapid evaluation of new concepts in HPC systems software, and it also enables the development of application or programming model specific environments.

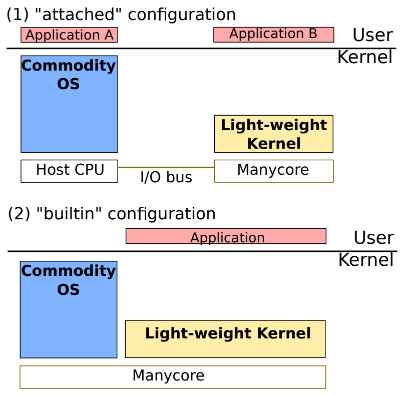


Figure 1. **Two configurations of manycore systems**

Interface for Heterogeneous Kernels, our proposed framework, aims to provide a general framework to enable hybrid kernel designs. IHK currently provides two types of configurations (Figure 1). The "attached" configuration supports manycore coprocessors attached to a multi-core host, while the "builtin" configuration allows partitioning resources (i.e., CPU cores and physical memory) of a standalone manycore platform.

III. IHK: Interface for Heterogeneous Kernels

A. Requirements

To accomplish the hybrid kernel approach described in the previous section, there are three requirements for IHK: (1) To manage multiple kernels and to provide an interface to allocate or free resources for a kernel, and boot or stop a kernel. (2) To partition resources in the node so that multiple kernels can coexist in the same node. (3) To provide a communication mechanism among the kernels in order to achieve integration of system services.

B. Basic Architecture

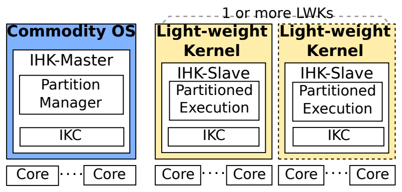


Figure 2. **Architectural overview of IHK components.**

The basic architecture of IHK is depicted in Figure 2. IHK categorizes kernels in two types: a master kernel and slave kernels. The master kernel is a kernel that is booted first through the normal booting process, for example, booted from BIOS or UEFI, and is typically a commodity operating system (such as Linux). Slave kernels are kernels that are booted from the master kernel. IHK's components in the master and slave kernels are called IHK-Master and IHK-Slave, respectively.

The features corresponding to the requirements are implemented as shown in Figure 2: resource management is implemented in IHK-Master, a feature to execute with the partitioned resources is implemented in IHK-Slave, and the communication facility called IHK-IKC is implemented both in IHK-Master and IHK-Slave.

There are two design policy in IHK. One is to provide a minimal interface to achieve hybrid kernel designs. Thus, the general functionalities in operating systems and should not be included in interface of IHK. One of the examples is to map the physical memory to the virtual memory, which is a common function in operating systems, but might not be necessary in a certain light-weight kernel. In contrast, IHK has a function to map the remote physical memory to the physical memory. It is a required feature to share some information among kernels, which is the case only when the heterogeneous kernel approach is taken. The other is to provide a utility library for kernels that is cumbersome but requisite. The example is the bootstrap code. It does not contradict the first policy because the library is opt-in for light-weight kernels, but rather it helps the rapid development of light-weight kernels.

IV. IHK Details

IHK consists of several interfaces and common libraries. The interface is exposed to each kernel so that the kernel can boot another one, it can coexist or communicate with other kernels. The common library provides an implementation for the part of the interface which makes it easier and quicker to develop new kernels running on top of IHK. In the following, details of each component in IHK is described.

A. IHK-Master

IHK-Master provides the interface which is used in a master kernel to boot another kernel in the same machine. Booting another kernel means that a certain set of CPU cores, a certain area of memory, and a certain set of devices, if any, are dedicated to the slave kernel. This requires management of resources and the ability to run multiple kernels within a single node.

1) Objects: The types of resources which IHK manages are "devices," "cores," and "memory." The resources are represented in the objects named `ihk_device`, `ihk_cpu_core`, and `ihk_memory_area`, respectively. A device is a multicore or manycore processor which has several cores and some amount of memory. If a kernel is booted from a host CPU in a manycore coprocessor, the target device would be the coprocessor. If a kernel is booted in another CPU core in the same SMP processor, the target device would be the processor itself. CPU cores and memory logically all belong to these devices.

The other object in IHK-Master is `ihk_kernel` which represents a running kernel. As we do not assume that a single kernel manages multiple devices, an `ihk_kernel` object has an associated device. Since a kernel requires dedicated CPU cores and memory to run, it has associated resources.

While the status of objects, e.g., whether resources are dedicated to some kernels or not, must be shared among the master kernels, the way of sharing depends on the implementation. In the current implementation, only one kernel serves as the master kernel, thus maintaining this information there is sufficient.

2) Functions: The master kernel boots another kernel by the following steps: the master kernel creates device objects, creates a new kernel object, assigns resources to the kernel, loads the kernel image, sets parameters for the kernel and finally boots the kernel. In the following, we describe the detailed steps and functions used.

When the device driver for a manycore device is loaded in the master kernel and the driver calls the `ihk_master_create_device` function, the master kernel creates an `ihk_device` object corresponding to the device. If the kernel is capable of booting another kernel in the same CPU, it creates an `ihk_device` object corresponding to the CPU. Each device is associated to a set of functions which handle device-specific actions such as accessing its memory or booting.

Booting another kernel is typically initiated by a user request, and the user interface is dependent on the master kernel implementation. As shown later in Section V-A, our Linux implementation uses character devices for this purpose. Upon request, the `ihk_master_create_kernel` function is called to create an `ihk_kernel` object.

Next, resources should be assigned to the kernel. There are two types of functions to do this: "allocate" and "reserve." In IHK-Master, the former only specifies the amount of resources required and lets the master kernel choose the specific set of resources, while the latter specifies which resource must be assigned to the kernel. Thus, for the CPU cores, the `ihk_master_allocate_cpu` function requires the number of CPU cores as the argument, while the `ihk_master_reserve_cpu` function requires the set of CPU cores specified by core IDs. As for physical memory, the argument of the `ihk_master_allocate_memory` function defines the amount of memory, and that of the `ihk_master_reserve_memory` function defines the start address and the size of the particular area.

Two functions are defined in IHK-Master to load the kernel image. The difference between the two functions is the source of the kernel image, i.e., whether it is a file or the image is in memory. To load from a file, the `ihk_master_load_file` function should be used; to load from a memory, the `ihk_master_load_image` function should be used. In the both cases, it loads the specified kernel image to a proper location in the assigned memory area.

Users may want to assign parameters to the new kernel, and this is achieved by via the `ihk_master_set_args` function. The argument is a simple string and the format is user specified.

Finally, the `ihk_master_boot` function initiates the slave kernel. The function kicks a CPU core allocated for the slave kernel so that it proceeds with booting.

The kernel status, whether the kernel is in the middle of booting, booted, in panic, or shut down, can be queried via the `ihk_master_query_kernel_status` function.

To stop a kernel, the `ihk_master_shutdown` function provides a way to shutdown the kernel safely.

When a slave kernel is stopped by some reason, including the `ihk_master_shutdown` function, the kernel object should be released by `ihk_master_destroy_kernel`. It forcefully stops the kernel if it is still running, releases the resources it has used, and releases the `ihk_kernel` object. This terminates the lifecycle of a slave kernel.

B. IHK-Slave

IHK-Slave defines interface for slave kernels to work with other kernels. As IHK-Master manages partitioning resources among the slave kernels, they only use resources dedicated to them by obtaining the resource information through IHK-Slave. The way IHK-Slave retrieves the information from IHK-Master is implementation-specific; typically, it retrieves by accessing a certain address in memory.

Three functions are defined in IHK-Slave. The first function, `ihk_slave_get_kernel_id`, retrieves the identifier to distinguish the running slave kernel. The second function, `ihk_slave_get_available_cpu`, returns the set of CPU cores which are available for the slave kernel. The third function, `ihk_slave_get_available_memory`, obtains the memory areas which are available for the kernel.

As partitioning must be assured by the slave kernels, the slave kernel should retrieve the resource partitioning information, that is, it should call the `ihk_slave_get_available_cpu/memory` functions during initialization phase of the booting. Under the assumptions that the master kernel works correctly, the latest time up to when the slave kernel is required to obtain resource information is as follows: As for CPU cores, the booted CPU core is guaranteed to be in the available CPU set, the function should be called before it wakes up any other CPU core. As for memory, the kernel code and statically allocated data (typically, the code and data located in the "text," "data" and "bss" sections) is guaranteed to be in the available area of memory, the function should be called before it starts dynamic allocation of memory.

C. IHK-IKC

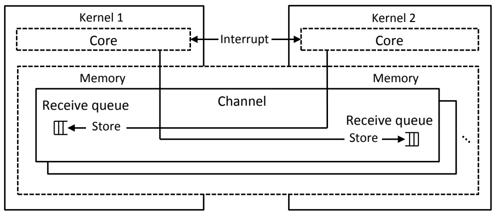


Figure 3. **Communication model in IHK-IKC.**

IHK-Master and IHK-Slave both have interface of IHK-IKC, message-passing peer-to-peer channel-based communication interface among multiple kernels. Figure 3 depicts the communication model of IHK-IKC. A channel consists of a pair of message queues where the size of a message is fixed. The sizes of a message and a queue can be configured for each channel. As IHK-IKC is designed for communication among kernels, the endpoints of a channel are identified by their kernel IDs and cores.

The core functions in IHK-IKC provide a basic set of ways to manipulate channels: initializing a channel (`ihk_ikc_create_channel`), destroying a channel (`ihk_ikc_free_channel`), receiving a message (`ihk_ikc_recv` and `ihk_ikc_recv_handler`), and sending a message (`ihk_ikc_send`). The reason why there are two receiving functions is to provide a function that avoids additional copying from a queue in the channel. The `ihk_ikc_recv_handler` function calls the specified handler function for every message received in the queue with the pointer to the message without copying. As it assumes that the receive queue is located in the local memory, its access cost would not excess. Dequeuing the message from the queue is deferred until the handler function returns, thus the handler function should be as light as possible. Alternatively, the `ihk_ikc_recv` function copies the message from the queue to the specified pointer.

IHK-IKC only defines the very primitive functions, and the other functionality in communication is left to implementation of these functions. For example, notification of arrival of a new packet to the other endpoint in the channel should be implemented in the `ihk_ikc_send` function; typically using interrupts or polling. IHK provides a library of IHK-IKC implementation for these functions with additional "listen and connect" port model and asynchronous message reception feature. The detail is shown later in Section IV-E.

D. Bootstrap Library

The initialization of a new core is highly architecture-dependent but common to every kernel running for the same architecture. IHK provides a "bootstrap library" for certain architecture to ease the development of an LWK. As our first target are Xeon and Xeon Phi, the current IHK has bootstrap implementation for the x86-64 architecture.

The bootstrap library has two components: the CPU initialization part, and the ELF boot part. The CPU initialization part initializes a CPU core. For example, it switches a CPU core to 16-bit real mode to 64-bit mode with paging enabled in x86-64 architecture. The ELF boot part loads an ELF kernel into memory according to its ELF header. This allows the light-weight kernel to employ the ELF format, a common binary format.

E. IKC Library

IKC Library provides an easier way to establish multiple channels in the server-client manner; the server endpoint "listens" to a certain "port," and the client endpoint "connects" to the port listened by the server. Channels are managed through a single channel called the "master" channel: e.g., the endpoints share the address of the queues in a channel via "master" channel packets. By passing control messages through the master channel, the IKC Library can establish and destroy channels.

When the IKC Library is initialized, the master channel is established between the master and the slave kernels. The "master" channel uses an arbitrary core for each kernel as its endpoint. The way to share the memory locations of the queues between the endpoint kernels is implementation dependent. When a kernel waits for a new connection in some port, it calls the `ihk_ikc_listen` function specifying the port number and a handler. When the peer kernel connects to the port by the `ihk_ikc_connect` function, IKC Library creates the channel and calls the handler with the information for the channel. In order to provide an asynchronous reception feature, channel are also associated with packet handlers which is set in `ihk_ikc_connect` or the listen handler. The packet handlers are called when a new packet arrives in the channel. The `ihk_ikc_disconnect` function disconnects the specified channel.

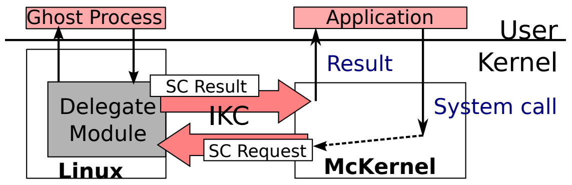


Figure 4. **Overview of the system call delegation mechanism.**

V. Implementation of Master and Slave Kernels

This section describes our implementation of IHK on Linux and a LWK. As a master kernel, Linux 2.6.38 is used and modified to support resource partitioning as implementing IHK-Slave. In Linux, character devices are also used for user interface for IHK-Master. As an example LWK, the implementation of McKernel, which is aimed to provide applications with a noiseless environment and the functionality of Linux, is then described.

A. *Linux: A master kernel*

The implementation of resource partitioning in Linux is based on the SHIMOS mechanism [15]. The SHIMOS mechanism is implemented by modifying and adding codes to Linux in order to limit use in Linux of CPU cores and memory, thus it achieves partitioning. SHIMOS manages the resource usage of kernels in a central structure which is referred and used in booting and allocating resources in IHK-Master.

The user interface for IHK-Master in Linux is implemented as VFS operations on the special character devices. The `ioctls` to the character directs IHK-Master to create, load, and boot the new kernels. These requests are then handled by IHK-Master, and in turn it calls the device-specific code for each request. The device-specific code is provided by a device driver kernel module which passes the function pointers for the device-specific handling routines to IHK-Master, therefore it is easily portable to the other manycore devices.

B. *McKernel: A Noiseless Kernel*

McKernel is designed to run applications without OS noises by having the minimalist kernel core and isolating it from various daemons, consequently reducing cache contentions caused by system calls and other processes. McKernel is equipped with a system call delegation mechanism to provide the same functionalities available on the master kernel to the applications running on McKernel. The system call delegation mechanism in McKernel is implemented on IHK-IKC, exemplifying one practical use it. In the following, we focus on the system call delegation mechanism in McKernel.

1) Structures: The structure of the system call delegation mechanism is illustrated in Figure 4. In Linux, there are a delegate kernel module that handles the IKC channels for system call delegation between McKernel, and “ghost processes” that performs system calls on behalf of the processes running in McKernel.

When a system call is issued in an application in McKernel, McKernel handles the system call if it is implemented in McKernel, and delegates it to Linux otherwise. The criteria if the system call is implemented or not in McKernel are whether it is performance-critical and whether it needs change the local processor state. The former example is “futex,” which are frequently called by multithread libraries for synchronization between threads; the latter example is “mmap” of anonymous pages, which requires page table manipulation for the processor. To delegate system calls, McKernel sends a message to Linux via a IKC channel.

One ghost process in Linux exists for one process in McKernel. The ghost process waits for system call requests from the corresponding process in McKernel. The delegate kernel module wakes up the corresponding ghost process when it receives a system call request via IKC, passing the information for the request to the ghost process. The ghost process executes a system call, and requests the delegate module to send the result to McKernel, and then waits for another system call requests.

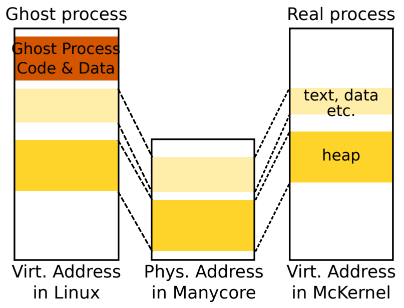
There arises a question for system calls: how ghost processes access the memory contents in McKernel and how the virtual addresses in the arguments are solved because the arguments may have pointers to the memory in McKernel and addressed by a virtual address in McKernel.

2) Handling Pointers in Delegated System Calls: The solutions for the question are (i) resolving the virtual addresses by McKernel before it sends system call requests to Linux or (ii) using the same virtual mapping in Linux as McKernel.

The first solution is that McKernel modifies the pointer arguments in system calls to the physical addresses. Linux maps the McKernel’s physical memory to its local virtual memory, and performs the system call by rewriting again the addresses of the physical pointer to the virtual pointer. Alternatively, Linux may copy the McKernel’s memory pointed by the pointer to its own buffer by using DMA engines if the mapping McKernel’s memory is not possible. However, in either case, it requires the knowledge of which arguments are pointers for all the system calls. It is sometimes very difficult for system calls that the semantics of arguments vary by the context, e.g. the `ioctl` system call.

The second solution is that a ghost process defines virtual to physical mappings in the same way as the corresponding real process, as illustrated in Figure 5. This unified address space layout allows the ghost process to access the memory area of the real process using the same virtual addresses. The code and data specific to the ghost process is mapped in an address range which is not used by the real process.

The benefit is that there is no need to recognize which arguments of the system calls are addresses, and what are the side effects of the system calls. However, the disadvantages are that accesses to the McKernel memory from Linux are



Ghost process
Real process
Ghost Process Code & Data
text, data etc.
heap
Virt. Address in Linux
Phys. Address in Manycore
Virt. Address in McKernel

Figure 5. **Unified virtual address space of the application and the corresponding ghost process.**

performed in the same way as the memory local to the ghost process, so it can result in the inefficient accesses like many random accesses to small memory areas over I/O bus, and that it cannot be used for accelerators that do not allow mapping of the memory in accelerators to the host.

McKernel employs the second solution because it can resolve the "ioctl" issues, and because the target manycore processor, Xeon Phi, is capable of mapping Xeon Phi's memory to the host address space.

VI. Light-Weight Kernel Examples

This section discusses various OS implementations on top of IHK with the intention of demonstrating IHK's ability to provide the basic foundation for rapid prototyping of new kernel designs.

A. *Hybrid Segmentation Kernel*

Our first example is a hybrid kernel design that leverages segmentation instead of paging for providing virtual memory targeting HPC and big data workloads [11]. Running a very lightweight kernel that sets the application CPU cores to segmentation mode and subsequently offloads kernel services to dedicated cores running a regular kernel on top of paging is the basic architecture proposed in [11].

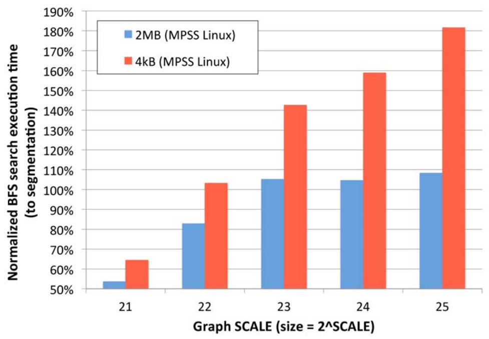


Figure 6. **Normalized average execution time of BFS search in Graph500 using 4kB/2MB pages on MPSS Linux compared to segmentation with McKernel as the function of graph scale [11].**

The segmentation configuration is evaluated using the Graph500 benchmark (Figure 6, and results show 81% and 9% improvement compared to utilizing 4kB and 2MB pages, respectively. IHK plays a crucial role in booting the minimalistic kernel and providing the infrastructure for system call offloading. Designing such a system from scratch, including the implementation of system calls in the segmentation kernel would have required significantly more development, without providing any additional value from the memory management point of view.

B. *Hierarchical Memory Management*

Another example of a specialized lightweight kernel has been introduced in [12] for dealing with hierarchical memory management in current heterogeneous architectures focusing on Intel's Xeon Phi manycore co-processor. In particular, this work introduces a novel page table arrangement called partially separated page tables (PSPT) which minimizes the cost of remote TLB invalidations when data movement is performed at the OS level and the address space of the application is often modified. PSPT uses separate page tables for each CPU core even if the corresponding threads run in the same address space which enables the OS to keep track of which virtual addresses are mapped by which cores. A 2D heat diffusion stencil computation benchmark was used to evaluate the performance of hierarchical memory management and results are shown in Figure 7. For more details on this study, refer to [12].

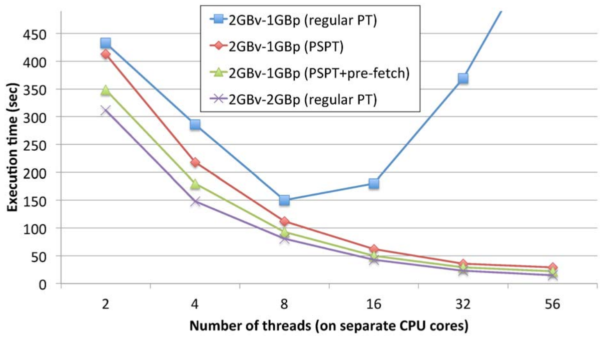


Figure 7. **Stencil computation over hierarchical memory [12].**

Again, with respect to IHK, it is important to point out that the development cost of the lightweight kernel running on the Xeon Phi was substantially lower than implementing such a system entirely from scratch. IHK provides the basic infrastructure for interaction between the Xeon Phi and the host machine and it also enabled rapid prototyping of page table level modifications which would be a major effort in the Linux kernel.

VII. Evaluation

In this section, we show the evaluation of the implementation of IHK in Linux and McKernel for x86_64 processor with Xeon Phi as manycore coprocessor. The evaluation environment is summarized in Table I. In the following results, "attached" and "builtin" denotes the configurations in IHK. In "attached," McKernel is running in Xeon Phi and

Table I
EVALUATION ENVIRONMENT

Host	CPU	Intel Xeon E5-2670 v2 x 2 (2.50GHz, 10 cores, 20 threads)
	Memory	DDR3 1600 MHz 64 GB
	OS	RedHat Enterprise Linux 6.2
	Compiler	Intel C/Fortran Compiler 13.0.0 20120731
Coprocessor	CPU	Intel Xeon Phi 5110P (1.053GHz, 60 cores, 240 threads)
	Memory	GDDR5 8GB
	OS (MPSS)	Intel MPSS 3.1.2

Table II
IKC AND SYSTEM CALLS LATENCY

	Attached	Builtin
IKC Ping-Pong	16.7 us	26.1 us
`getuid`	21.5 us	39.2 us

Linux is running in the host machine. In "builtin," Linux and Mckernel are both running in Xeon Phi, and Linux is also running in the host machine. "MPSS-Linux" denotes Intel's default configuration that runs one Linux kernel in Xeon Phi, and also Linux in the host machine.

In this section, the performance of IHK-IKC, and the system call latency implemented upon IHK-IKC in McKernel are shown. Finally, as the goal of McKernel is to provide a noiseless environment for applications, the application performance and variance in McKernel is presented.

A. *IHK-IKC*

To evaluate the performance of IHK-IKC, we implemented a simple IKC channel that just performs message ping-pongs. The round-trip time for a message ping-pong between Linux and McKernel was measured. To discuss the latency, the latency for system call implemented upon IHK-IKC in McKernel was also measured. The used system call is the `getuid` system call, which just loads and returns a value in the task structure. The results are shown in Table II. The system call latency in the "attached" case is about 22us, which includes costs for waking up ghost processes and system call procedures, as shown in Figure 4.

Next, the performance of I/O system calls was measured. The measurement is conducted to the file in a *tmpfs* file system. For the "attached" case, the file is located in the host, for the "MPSS-NFS" case, the file is located in the host mounted by NFS in Xeon Phi via a virtual network device provided by MPSS. and for the other cases, the file is located in Xeon Phi. The data size used was 64MB. Table III shows the results. The system call delegation overhead is negligible when more than one megabyte is read because the 22us overhead occupies only 7.8% of the latency of one megabyte read for "attached." Moreover, from the comparison between "MPSS-Linux" and "attached," the I/O performance is better when delegated to the host despite of the delegation cost. The "MPSS-NFS" case does almost the same as "attached," but it uses NFS over virtual network devices, the performance was worse than "attached" This implies that the higher integration into systems service gets better performance for the kernel communication.

Table III
I/O SYSTEM CALL PERFORMANCE

	Attached	MPSS-NFS	Builtin	MPSS-Linux
Read	3543 MB/s	439 MB/s	404 MB/s	411 MB/s
Write	1131 MB/s	306 MB/s	384 MB/s	392 MB/s

Table IV
ELAPSED TIME (SECOND) OF NAS PARALLEL BENCHMARK IN (# OF THREADS = 59)

Name	McKernel			MPSS-Linux		
	Mean	$+2\sigma$	Worst	Mean	$+2\sigma$	Worst
BT.B	47.58	47.59	47.59	50.77	51.20	50.96
CG.B	49.48	49.49	49.49	52.73	54.16	53.53
EP.C	27.77	27.88	27.85	29.54	30.05	29.69
FT.B	9.48	9.48	9.48	9.51	9.94	9.81
IS.C	3.33	3.33	3.33	3.50	3.60	3.53
LU.B	55.71	55.71	55.71	58.86	59.10	58.98
MG.B	1.33	1.35	1.35	1.06	1.12	1.10
SP.B	50.28	50.33	50.32	50.80	53.50	52.29

B. *McKernel*

The goal of McKernel is to prevent applications from suffering from the noises in the single commodity operating systems. Next, we ran the OpenMP version of NAS Parallel Benchmarks[16] to evaluate the performance of applications and the variability of performance. We ran the benchmark in the "attached" environment and "MPSS-Linux" environment. The classes (problem size) for the benchmarks were chosen according to their memory consumption to fit into the memory of Xeon Phi. Because OpenMP uses a thread to monitor the threads in addition to the worker threads, the benchmarks were executed with a configuration of 59 calculation threads. The results of repeating each benchmark ten times is shown in Table IV. The table shows the mean, the 95th percentile ($+2\sigma$), and the worst of the results for each benchmark. The result shows that the mean benchmark performance are slightly better than Linux with an exception, *MG.B*. It needs further investigation for the reason, but it is considered to be due to insufficient optimization on thread synchronization mechanism, such as `futex` system calls. On the other hand, Linux has more variability in performance when we compare the mean values and 95th percentile values. This stable behavior in performance in McKernel will have more impact when the applications are executed over multiple nodes.

VIII. RELATED WORK

This section describes related work in the view of lightweight kernels, inter-kernel interface.

A. *Kernel Architectures*

In this section, we select several research efforts related to IHK and the hybrid kernel design.

L4Linux[17] runs Linux on top of the L4 microkernel. By building up two types of kernels, it is aimed to be a practical operating system with the flexibility of the microkernel. The approach is integrating of heterogeneous kernels rather than coexisting and cooperating, therefore required a lot of modifications on Linux. The modification of SHIMOS-Linux is almost limited to the code around the booting and it does not alter the core kernel functionality.

Exokernel [18] provides a minimalistic kernel-space, leaving the other kernel functionalities implemented in user-space as library OS (LibOS). As LibOS is a user-space library, it can be specialized for processes which it links to. This allows unified design of system services to applications, which is similar to the goal of the hybrid kernel design. However, the hybrid kernel design achieved by IHK enables changing the whole operating system service in the privileged mode, allowing wider range of ideas in systems software to be implemented than in the user-space; for example, segmentation kernel.

ZeptoOS [19] tries to make Linux perform as better as the default light-weight kernel (CNK) in IBM Blue Gene supercomputers by eliminating the source of performance degradation issues in Linux. The approach is the opposite of the light-weight kernel approach, and applies the performance improvement to the existing fat operating systems. This approach might be effective for a single generic goal, but to test various radical ideas in operating systems, changing the large existing operating systems would be too costly.

B. Hybrid kernels

FusedOS [9] combines FWK (full-weight kernel) and LWK (light-weight kernel). It provides applications with rich functionality by FWK and noiseless environment by LWK. However, it lacks abstraction interface like IHK that enables different types of kernels, not only Linux and McKernel, to be used. IHK provides more general way to accomplish hybrid operating systems environments. In addition, FusedOS uses user-level layer for LWK functionality, delegating all the system calls to FWK, which results in performance degradation when applications invoke much system calls. One of the most frequently used system call in parallel programs is “futex.” McKernel provides both system call delegation mechanism and system call handlers. Thanks to this design, McKernel can handle performance-critical system calls including “futex” by itself and can avoid performance degradation.

C. Inter-kernel interface

The interface for GPUs and accelerators are proposed in several projects including OpenCL[20], and PTask[21]. They are aimed for parallel computation by user processes, not for kernel extension used by operating system kernels. IHK assumes that the manycore coprocessor can execute operating system kernels, but for GPUs, it is not the case. For such accelerators, the application-level interface such as OpenCL and PTask is appropriate.

Helios[22] provides a single interface of managing heterogeneous systems, especially systems with programmable devices. It offloads some of the kernel functions to the programmable devices via the interface, and achieves higher performance.

The idea to make system calls offloaded or concentrated to certain cores as adopted in FusedOS and McKernel is already proposed in several existing researches, including FlexSC [23] and GenerOS [24]. However, the communication used to achieve the ideas is not generalized, but just implemented for the purpose of system call delegation. For those operating systems, it is difficult to implement the other features additionally that require communication between kernels.

MCAPI[25] defines the communication API between cores in multicore systems, especially for embedded systems. IHK includes communication API set between kernels in manycore systems. However, IHK also defines APIs to manage different kernels, and integrates the required interface including these two APIs. The goal of IHK is a general framework to accomplish hybrid kernel designs by providing all the APIs.

IX. Conclusion

Commodity operating systems have various issues to overcome so that they can be deployed on upcoming manycore systems. Hybrid kernel designs are now considered to be a promising approach for exascale. In this paper, we have proposed Interface for Heterogeneous Kernels (IHK) that provides a general framework to build such hybrid kernel systems. Certain types of applications may benefit from a particular idea of new kernel implementation. The framework provided by IHK enables quick implementation and evaluation of such ideas. We believe that rapid evaluation is mandatory when system software needs to be highly integrated with applications and/or with special features of upcoming exascale hardware. To demonstrate IHK’s ability for rapid prototyping, we have presented McKernel, a hybrid segmentation kernel and a hierarchal memory management system, all implemented on top of IHK.

In the future, we will extend IHK to I/O functionality. A direct communication facility with other nodes in McKernel discussed in [26] makes InfiniBand interconnect devices directly accessible from McKernel running on the Xeon Phi. It would be beneficial to provide a generalized I/O interface via IHK to various OS kernels.

Acknowledgement

This work is partially supported by feasibility study on advanced and efficient latency core-based architecture for future HPCI R&D funded by the Ministry of Education, Culture, Sports, Science and Technology, Japan. This work is also partially supported by a CREST project of Japan Science and Technology Agency (JST).

References

[1] T. Shimosawa, “Operating System Organization for Manycore Systems,” dissertation, The University of Tokyo, 2012.

[2] Intel Corporation, “Intel®Xeon Phi™Product Family,” http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] TOP500.org, “TOP 500 Supercomputing Sites - November 2013,” http://www.top500.org/lists/2013/11.

[4] R. Hazra and B. Davis, “Technical Computing - Discover Your Parallel Universe,” http://newsroom.intel.com/servlet/JiveServlet/download/38-28204/SC%2713_Intel_presentation.pdf.

[5] F. Petrini, D. J. Kerbyson, and S. Pakin, “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q,” in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, p. 55.

[6] An Exascale Operating System and Runtime Research Project, “Argo: An exascale operating system,” http://www.argo-osr.org/.

[7] S. M. Kelly and R. Brightwell, “Software Architecture of the Light Weight Kernel, Catamount,” in *Proceedings of the 2005 Cray User Group Annual Technical Conference*.

[8] J. Moreira, M. Brutman, J. Castaños, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt, “Designing a highly-scalable operating system: the Blue Gene/L story,” in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 118.

[9] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. Wisniewski, “FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment,” in *IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2012)*, Oct 2012, pp. 211–218.

[10] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, “mOS: An architecture for extreme-scale operating systems,” in *The 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*, 2014, pp. 2:1–2:8.

[11] Y. Soma, B. Gerofi, and Y. Ishikawa, “Revisiting virtual memory for high performance computing on manycore architectures: A hybrid segmentation kernel approach,” in *The 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '14)*, 2014, pp. 3:1–3:8.

[12] B. Gerofi, A. Shimada, A. Hori, and Y. Ishikawa, “Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures,” in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2013, pp. 360–368.

[13] ARM Limited., “big.LITTLE Technology: The Future of Mobile,” http://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf.

[14] C. Sosa and B. Knudson, *IBM System Blue Gene Solution: Blue Gene/P Application Development*, 2009.

[15] T. Shimosawa, H. Matsuba, and Y. Ishikawa, “Logical partitioning without architectural supports,” in *IEEE International Computer Software and Applications Conference*, 2008, pp. 355–364.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, “The NAS parallel benchmarks – summary and preliminary results,” in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 158–165.

[17] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, “The performance of μ-kernel-based systems,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pp. 66–77.

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., “Exokernel: an operating system architecture for application-level resource management,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, Dec. 1995, pp. 251–266.

[19] K. Yoshii, K. Iskra, H. Naik, P. Beckman, and P. C. Broekema, “Performance and Scalability Evaluation of ‘Big Memory’ on Blue Gene Linux,” *International Journal of High Performance Computing Applications*, vol. 25, no. 2, pp. 148–160, 2011.

[20] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science and Engineering*, vol. 12, pp. 66–73, 2010.

[21] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “PTask: operating system abstractions to manage GPUs as compute devices,” in *The ACM SIGOPS 23rd symposium on Operating systems principles (SOSP '11)*, pp. 233–248.

[22] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, “Helios: heterogeneous multiprocessing with satellite kernels,” in *The ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pp. 221–234.

[23] L. Soares and M. Stumm, “FlexSC: flexible system call scheduling with exception-less system calls,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI '10)*, 2010, pp. 1–8.

[24] Q. Yuan, J. Zhao, M. Chen, and N. Sun, “GenerOS: An asymmetric operating system kernel for multi-core systems,” in *IEEE International Symposium on Parallel Distributed Processing (IPDPS 2010)*, 2010, pp. 1 –10.

[25] The Multicore Association, “MULTICORE COMMUNICATIONS API WORKING GROUP (MCAPI(R)),” http://www.multicore-association.org/workgroup/mcapi.php.

[26] M. Si, Y. Ishikawa, and M. Takagi, “Direct MPI Library for Intel Xeon Phi Co-Processors,” in *IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2013, pp. 816–824.