IPSJ SIG Technical Report

Vol.2015-ARC-215 No.1
Vol.2015-OS-133 No.1
2015/5/26

Operating System Design for Next Generation Many-core based Supercomputers

Masamichi Takagi[1,a) Balazs Gerofi[1,b) Norio Yamaguchi[1,c)
Takahiro Ogura[1,d) Toyohisa Kameyama[1,e) Atsushi Hori[1,f)
Yutaka Ishikawa[1,g)

Abstract: Processor core count in high-end computing has seen a steady increase during the past decade and next generation supercomputers will likely deploy many-core based systems. At the same time, from a software environment point of view, Linux-compatibility has become wide-spread in the High Performance Computing (HPC) domain. We consider the challenges of operating system (OS) design targeting next generation high-end computing. We believe that the most urging issues to be addressed are as follows. (1) Exploiting deep memory hierarchies, (2) Reducing cache pollution by OS services and minimizing OS noise, (3) Making it easy to design and deploy application specific kernels, (4) Providing a Linux compatible programming / run-time environment and (5) Enabling seamless tracking of upstream Linux kernel changes. We contend that existing approaches to HPC operating systems, which either employ a stripped down Linux environment or a specific light-weight kernel built from scratch, are not feasible to deal with these challenges. In this paper, we discuss the design decisions of our proposed hybrid kernel design for providing a Linux-compatible light-weight kernel.

1. Introduction

Next generation supercomputers should embrace and exploit technological advances of the recent past and the future. For example, it is anticipated that in future systems a new layer in the memory hierarchy will be added to processor architectures, and the number of CPU cores applications can use will reach the order of a million. Application developers have been increasingly relying on the standard Linux APIs, but at the same time, application specific kernels have been proven effective [1].

The most important challenges for OS design considering these requirements are summarized as follows.

- **Exploiting deep memory hierarchies** Processors will have multiple memory types with different characteristics. The kernel should optimize the allocation of kernel data structure to those memory-areas.
- **Reducing cache pollution by OS services and minimizing OS noise** Kernel code should minimize pollution of processor caches used by application code and it should not waste valuable CPU cycles of the application. Most importantly, OS kernels for high-end computing should ensure scalable application performance.
- **Facilitating deployment of application specific kernels** Applications may benefit from customized OS kernels tailored for specific application needs and users should have the freedom to choose the appropriate kernel at job dispatching time.
- **Linux compatibility** A Linux compatible programming and run-time environment should be provided because users have been increasingly relying on the Linux APIs in the HPC domain as well.
- **Maintainability** An OS using a configuration of running a lightweight-kernel alongside with Linux kernel and interacting with it should track Linux kernel changes to incorporate its improvements. However, careful design on which Linux modules to reuse or interact with is required because more reuse and interaction means more code we should modify when Linux kernel changes.

Currently, there are two major approaches to operating systems on high-end supercomputers. The first one is eliminate features a Full-Weight Kernel (FWK), often Linux, that prevent scalable application performance. The second one is to implement a Light-Weight Kernel (LWK) from scratch with minimal functions to provide an environment tailored for HPC. However, existing work cannot deal with all of the above mentioned challenges. For example, it is not easy for

1 RIKEN AICS
a) masamichi.takagi@riken.jp
b) bgerofi@riken.jp
c) norio.yamaguchi@riken.jp
d) t-ogura@riken.jp
e) kameyama@riken.jp
f) ahori@riken.jp
g) yutaka.ishikawa@riken.jp

the FWK approach to provide users a choice of application specific kernels. This is because it is difficult for kernel developers to create their own kernels since it is often the case FWK's concepts and representations ruling the code structure rarely matches those of the developers and they need to modify a large part of FWK. On the other hand, the LWK approach fails to provide the level of Linux compatibility that would be required so that the same tool-chains or executables could be run.

We have already proposed a new design of OS for next generation many-core based supercomputers in response to these requirements [2]. The kernel called McKernel is an LWK which runs alongside with Linux kernel and communicates with it using a library/framework called Interface for Heterogeneous Kernels (IHK). McKernel implements OS services critical to application performance, e.g., process management, memory management, signal management, critical system calls, and others are delegated to Linux via IHK. In this way, it can provide application with a Linux-compatible environment and at the same time it still reduces cache pollution and OS noise.

The design will be reviewed and the design issues obtained on the course of its development is discussed in this paper. The rest of the paper is organized as follows. Section 2 explains the background, Section 4 explains the related work, Section 3 describes the design of the kernel and Section 5 concludes the paper.

2. Background

2.1 Environment

Trend in supercomputer environment are explained in more detail.

2.1.1 Node Architecture and System Architecture

There are three major changes that are expected to happen in node/system architecture. The first one is that the number of nodes in a system is increasing so that the number of cores are reaching one million. The second one is a processor will have more layers in memory hierarchy. For example, Intel Xeon Phi, whose code name is Knights Landing will have stacked DRAM modules (called MCDRAM) with high bandwidth in addition to the traditional DRAM modules [3]. The third one is a many-core processor will have many NUMA domains.

2.1.2 Application Specific Kernels

It is beneficial for an application to select a set of OS functionalities that provide better performance to the application. For example, paging can be replaced with segmentation to eliminate TLB-miss and page-fault penalty[1]. Another example is to turn off time-sharing and timer interrupt to reduce OS noise for applications that do not oversubscribed threads which is often the case in HPC domain.

2.1.3 Programming Environment

Users of HPC domain rely on Linux environment. For example, 97% of machines in TOP500 list in November 2014 uses Linux [4].

2.2 Challenges

Challenges derived from the changes in the environment are explained in more detail.

2.2.1 Exploiting On-chip Memory Hierarchy

Kernel should optimize data allocation as in the same way as in multithreaded application. That is, kernel allocates the thread structures and page tables for the application thread to the NUMA-node on which the application thread is running to reduce costly data transfer between NUMA-nodes.

2.2.2 Cache pollution and OS noises

OS service processing should not compete with application threads for both hardware and software resources shared with OS and applications. For example, application would run hundreds of thread in parallel and a large part of data set referred for calculation resides on the cache of the individual cores. OS service processing could steal the cache resource, resulting in performance degradation of application. Shimosawa evaluated the performance degradation by cache pollution on Knights Ferry processor of Intel[5]. Xeon Phi processor of the current generation experiences a small amount of performance degradation and this issue is not a major one. However, the new processor architecture will introduce more NUMA-nodes and it is possible that this issue becomes more prominent. There are three methods to reduce cache pollution.

(1) Do not run processes other than application processes, e.g. system processes, on the core on which the application processes are running.
(2) Do not perform device interrupt processing on such kind of cores.
(3) Do not perform system call processing on such kind of cores.

Linux can employ the first two methods using cgroups.

OS noise should be reduced as well. OS noise refers to the state that OS services, e.g. periodic file writes of daemon processes and interrupt processing, utilizes processor resources and it can degrade performance of massively parallel applications. For example, assume an application using Bulk Synchronous Parallel (BSP) model. Its processes repeatedly synchronize in phases. The amount of OS noises changes in time and space, which makes some processes take longer time for the processing of a phase than others, which in turn make others wait on synchronization.

OS noise can be reduced by dividing cores to the cores for application and the cores for OS services and assigning daemon processes and interrupt processing to the OS service cores. However, timer interrupt handling and process scheduling code remains on the application cores and it is not possible to eliminate all of OS noises. It is possible to eliminate the last part by making applications not create more threads than the number of cores or making applications perform thread management.

2.2.3 Linux compatibility

Linux compatible environment should be provided when taking an LWK approach. This is because many of the applications on supercomputers are developed on Linux. There

Vol.2015-ARC-215 No.1
Vol.2015-OS-133 No.1
2015/5/26

are four major functionalities to provide Linux compatible environment. The first one is to provide Linux compatible system calls. For example, applications written in Fortran, C, C++ languages utilize `open`, `read`, `write`, `mmap`, `clone` and `sched_yield` system calls. The second one is to provide Linux compatible /proc and /sys file systems. /proc and /sys file system is a special file system which provide access to the system properties. For example, Intel OpenMP runtime and gdb rely on those file systems. The third one is to provide a job execution environment in a way that there is no need for users to distinguish in terms of programming environment what kind of kernel they are using or to modify their job-scripts according to the kernel type. The fourth one is to enable users to choose which kind of cores, that is, application cores or OS cores, to use for preprocessing part of a job. For example, current McKernel provides poor performance for `fork/exec` and hence preprocessing using `fork/exec` runs efficiently on OS cores (i.e. Linux cores). Another example is that preprocessing using `fork/exec` system calls might require process scheduling which McKernel does not offer and hence it runs only on OS cores. The last two functionalities are discussed in [6] and not discussed in this paper.

2.2.4 Facilitating creation of application specific kernel

Code complexity should be reduced to facilitate creating application specific kernel. One way to deal with this is to take an LWK approach and create a new kernel from scratch. The code complexity can be reduced if many OS services can be delegated to Linux side.

2.2.5 Maintainability

The LWK code need to be modified when Linux kernel changes and the cost should be minimized. This is because Linux compatibility is often provided by interacting with Linux and LWK should be modified when the way of the interaction changes. For example, an LWK which runs alongside with Linux can provide Linux-compatible /proc file system by reusing an infrastructure provided by Linux and redirect accesses to a sub-set of /proc files to files serviced by Linux. However, the infrastructure or the sub-set changes as Linux kernel changes and LWK side should keep up with the change. And the amount of change depends on the design on how much code to reuse and how deep the interaction is. Therefore, design on providing Linux compatibility should be done with those aspects took into consideration.

3. Design

3.1 Dynamic Resource Partitioning

McKernel relies on a low-level software infrastructure called Interface for Heterogeneous Kernels (IHK). IHK enables partitioning node resources and the management of lightweight kernels on subsets of the resources and it provides a low-level inter-kernel communication (IKC) framework. Figure 1 shows the architecture. A software module embodied as a kernel module, called IHK-master, resides on the Linux side and a software module embodied as a sub-

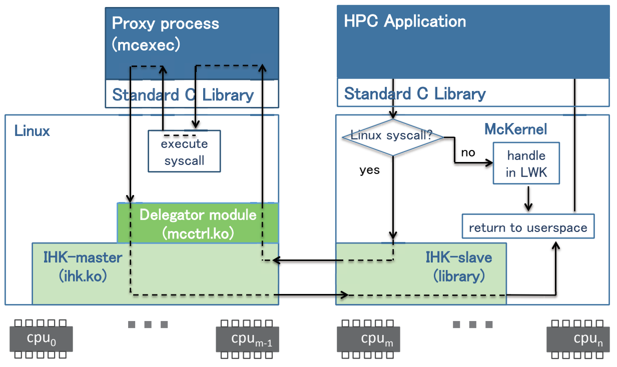

Fig. 1 Overview of the architecture.

module of McKernel, called IHK-slave, resides on the McKernel side. They communicate each other by using IKC.

Further detail of the IHK-master is explained. IHK-master consists of two types of modules. *IHK-master core* provides the basic IHK framework and management infrastructure. It is required for registering/removing the so called *IHK-master drivers*. *IHK-master drivers* represent resources, such as CPU cores and the physical memory of a given node or PCI-Express attached co-processors. Our most recent IHK driver module, *IHK-SMP x86* exposes a virtual device that enables partitioning CPU cores of an x86 (Xeon) SMP chip as well as the physical memory attached to the node among OS instances in a dynamic fashion. IHK-SMP relies on the CPU hotplugging system of the Linux kernel and it is also capable of handling NUMA specific dynamical memory allocation and assignment.

3.2 Tracking Linux Changes

As we outlined above, taking either path of integration between Linux and an LWK it is highly desired to keep Linux changes minimal. The Linux code base is a rapidly evolving target and keeping patches up-to-date with the latest kernel changes can be a major development effort.

IHK and McKernel requires no changes to Linux and is implemented in the form of a collection of stand-alone kernel modules. However, IHK-SMP relies on accessing a couple of unexported Linux kernel symbols (via the *System.map* symbol file) for resource partitioning. While this is not the intended usage of kernel modules in general, the Linux community seems to accept it and modules with similar mechanisms (e.g., the BLCR checkpoint/restart library [7]) are part of major Linux distributions.

3.3 Proxy Model and System Call Delegation

One of the crucial points of the proposed hybrid kernel configuration is the integration method between Linux and the LWK. Ideally, the application running on McKernel should not experience any difference in terms of Linux API availability, regardless what is supported by McKernel natively.

There are multiple possible approaches how the desired symbiosis may be attained and the one followed by McKernel is discussed here. We call this method the proxy model.

The central idea of the proxy model is that for each application executed on the LWK, a corresponding proxy process (also referred to as *ghost process*) on the Linux side is created. This architecture is shown in Figure 1. Five modules are involved in the proxy model. The first one is the application itself. The second one is the IHK-slave in McKernel. The third one is the IHK-master in Linux. The fourth one is a kernel module called `mcctrl.ko` in the Linux side. The fifth one is the proxy process.

Because McKernel implements only a subset of the Linux services, i.e., the performance sensitive system calls, the rest of the OS services need to be executed on Linux. Essentially, the proxy process provides the execution context on behalf of the application (running on the LWK) so that the offloaded calls can be directly invoked. The proxy process also ensures that Linux maintains certain state information that would have to be otherwise kept track of in the LWK. For example, McKernel has no notion of file descriptors, but rather it simply returns the file descriptor number it receives from the proxy process when a file is opened. The actual set of open files (i.e., file descriptor table, file positions, etc..) are managed by the Linux kernel. On the other hand, maintaining state in Linux implies that a certain degree of synchronization between McKernel and the Linux state, e.g., the unified address space described below.

With respect to system calls, our approach is that McKernel provides native support only for a minimal set of kernel features, the ones that are either performance critical or change the local processor's state or extentions done by McKernel. It has its own memory management, it supports processes and multi-threading with a simple round-robin cooperative scheduler, and it implements signaling. It also allows inter-process memory mappings and it provides interfaces to hardware performance counters. The minimal configuration of McKernel has no native support for disk device drivers, file systems, etc., and all these services are available with the help of Linux.

Every system call not provided natively get offloaded to Linux. In Linux, as part of IHK (See Section 3.1), a delegator kernel module (`mcctrl.ko`) handles IKC channels for system call delegation between McKernel and the proxy process that performs the calls on behalf of the actual application. During system call delegation, McKernel sends a message to Linux via a dedicated IKC channel. As mentioned earlier, for each application on the LWK, a corresponding proxy process exists in Linux. The proxy process waits for system call request messages via an ioctl() call into the delegator kernel module. The delegator kernel module's IKC interrupt handler wakes up the ghost process when it receives a system call request message, passing the information necessary to execute the system call (i.e., system call number and its arguments). The ghost process then executes the system call and requests the delegator module to send the result back to McKernel, which simply passes the return value to user-space.

A problem arises, however, as to how the ghost process on Linux can access the memory of the application running on McKernel and how the virtual addresses in arguments can be resolved. The problem arises because certain system call arguments may be just pointers (e.g., the buffer argument of a read() system call).

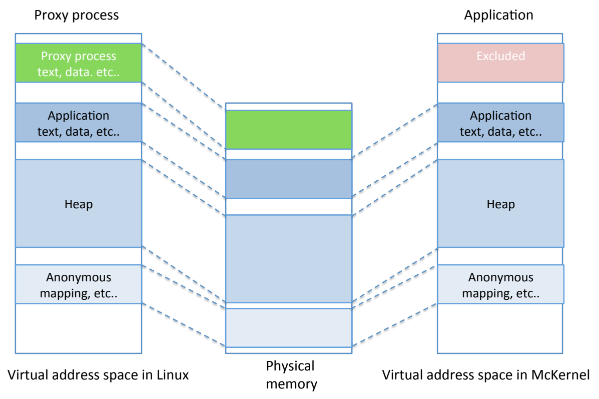
Proxy process
Application
Proxy process text, data. etc..
Excluded
Application text, data, etc..
Application text, data, etc..
Heap
Heap
Anonymous mapping, etc..
Anonymous mapping, etc..
Virtual address space in Linux
Physical memory
Virtual address space in McKernel

Fig. 2 Unified virtual address space of the proxy process on Linux and the corresponding application on McKernel.

McKernel's solution for the pointer issue is that the proxy process employs the same virtual to physical mappings as the actual application, as illustrated in Figure 2. This so-called *unified address space layout* allows the ghost process to access the memory area of the application using the same virtual addresses. The code and data segments specific to the proxy process are mapped in an address range which is explicitly excluded from McKernel's user space region.

The benefit is that there is no need to recognize which arguments of a system call are addresses. Moreover, any side effects of a system call (e.g., modifications to user-space data carried out by the Linux kernel) can naturally proceed.

The proxy does not need to know in advance which virtual address is mapped to which physical page. This is because Linux uses a special pseudo file mapping that covers the entire McKernel user space virtual address range, and every time an unmapped address is accessed, the page fault handler consults the page tables corresponding to the application on McKernel. As mentioned above, this requires that the mappings are occasionally synchronized, for instance, when the application calls `munmap()` or modifies certain mappings.

3.4 Specialized Lightweight Kernels

We contend that providing application or hardware specific lightweight kernels can benefit application performance due to fine grained tuning of related kernel services. For example, certain HPC applications may oversubscribed the node with more threads than CPU cores in order to exploit asynchronous operations, requiring the OS to support time sharing. Other applications may not need support for such configuration, leaving space for simplifying and optimizing process management in the kernel. Similarly, different lightweight kernels may be developed for supporting specific hardware features, e.g., management of memory hierarchies or heterogeneous core architectures.

Taking these disparities into account, there is no single set of kernel characteristics that could fulfill all the possible application and/or hardware requirements. To this end, IHK enables repartitioning node resources dynamically and allows loading and booting different lightweight kernel images through its easy LWK reboot feature. In fact, in our earlier work, we already demonstrated how applications may benefit from such optimizations [1], [8], [9].

3.5 Linux-compatible environments

3.5.1 Linux-compatible system calls

McKernel provides Linux-compatible system calls by delegating some of them to Linux and servicing others inside McKernel for the following two purposes. (1) Keep McKernel code minimal to make it easy to customize / extend its functionalities and (2) reduce cache pollution and OS noise. System calls meet the following criteria are processed in McKernel and others are delegated to Linux.

(1) Application performance can be improved by processing inside McKernel
(2) There is no way other than to process inside McKernel

The system calls which meet the first criteria are categorized as follows.

- **Process management functions** Creating thread (`clone()`), synchronization (`futex()`), setting CPU affinity (`sched_setaffinity()`)
- **Memory management functions** Allocating memory area (`mmap()`), duplicating memory area (`fork()`) and sharing memory area (`process_vm_readv`)
- **Profiling functions** Obtaining performance counters (`gettimeofday()` and PAPI interface)

The system calls which meet the second criteria are categorized as follows.

- **Process management functions** Setting signal handlers (`sigaction()`)
- **Debugging functions** Tracing and manipulating process (`ptrace()`)

System calls serviced by McKernel itself are show in **Tables 1, 2, 3 and 4**. All other system calls not shown in these tables are delegated to Linux. The system calls serviced by McKernel would change when Linux API semantics changes.

There is a special case that Linux and McKernel perform memory management in a cooperative way with the help of IHK. That is, assume a process running on McKernel using InfiniBand (IB) Host Channel Adapter (HCA). The McKernel process instructs the kernel module, which is running on Linux and the part of IB HCA driver, to pin-down a memory-area of the McKernel process via Linux-kernel internal function so that the memory area can be transferred from/to the HCA via DMA. IHK gives Linux a view of all physical memory through `struct page` at the partitioning time to allow the kernel module to perform such an operation.

Table 1 System calls serviced by McKernel itself (process management)

Implemented	Planned
arch_prctl	get_thread_area
clone	getrlimit
execve	ptrace
exit	rt_sigtimedwait
exit_group	set_thread_area
futex	setrlimit
getpid	signalfd
getrlimit[a]	signalfd4
kill	
pause	
ptrace2	
rt_sigaction	
rt_sigpending	
rt_sigprocmask	
rt_sigqueueinfo	
rt_sigreturn	
rt_sigsuspend	
set_tid_address	
setpgid	
sigaltstack	
tgkill	
vfork	
wait4	

[a] Some functions have not been implemented.

Table 2 System calls serviced by McKernel itself (memory management)

Implemented	Planned
brk	get_robust_list
gettid	mincore
madvise	mlockall
mlock	modify_ldt
mmap	munlockall
mprotect	set_robust_list
mremap	shmat
munlock	shmctl
munmap	shmdt
remap_file_pages	shmget
	process_vm_readv
	process_vm_writev

Table 3 System calls serviced by McKernel itself (scheduling)

Implemented	Planned
sched_getaffinity	alarm[b]
sched_setaffinity	getitimer[b]
	gettimeofday[b]
	nanosleep[b]
	sched_yield
	setitimer[b]
	settimeofday[b]
	time[b]
	times[b]

[b] These system calls are delegated to Linux for the moment.

3.5.2 Linux-compatible /proc and /sys file systems

McKernel provides Linux-compatible /proc and /sys file systems by reusing some files served by Linux and servicing others inside McKernel. Files of /proc and /sys file systems are added by kernel modules through interfaces given by Linux, with corresponding call-back functions which is called

Table 4 System calls serviced by McKernel itself (performance counters)

Implemented	Planned

Original Interface
`pmc_init`
`pmc_reset`
`pmc_start`
`pmc_stop`
	PAPI Interface

Table 5 /proc and /sys files or directories serviced by McKernel itself

File / Directory	Description
/proc/stat	Process statistics
/proc/<PID>/tasks/<TID>/stat	Thread statistics
/proc/<PID>/mem	Process memory contents
/proc/<PID>/task/<TID>/mem	Thread memory contents
/proc/<PID>/aux	Auxiliary vectors put in stack
/proc/<PID>/pagemap	Virtual-physical map

when access to them occur, and hence McKernel uses this infrastructure. McKernel needs to provide files representing the system properties to its processes but some of the files can be obtained from Linux-provided /proc and /sys file systems because some of the system properties do not change through resource partitioning and running McKernel alongside with Linux. Therefore, McKernel emulates a file system where accesses to some files are redirected to files served by McKernel and accesses to other files just see the files served by Linux. The examples of the files served by McKernel are (1) the memory maps of McKernel processes and (2) the number of cores in the McKernel partition. Accesses to files served by McKernel is served using IKC because the call-back function is called on the Linux side. Files and directories served by McKernel and implemented is shown in **Table 5**. We are investigating files which need to be served by McKernel.

3.5.2.1 Steps

Files served by McKernel are created at the boot time of McKernel and the boot time of McKernel thread.

File view where both files served by McKernel and files served by Linux coexist are performed using the following steps.

(1) Create files served by McKernel on /proc file system. Use the paths of `/proc/mcos<OS number>/<path>` for files to be shown as `/proc/<path>` to McKernel processes.
(2) Check the path when `open()` system call is delegated to `mcexec` and `mcctrl`.
(3) If the path has the prefix of `/proc/`, then replace the prefix with `/proc/mcos<OS number>/` and then perform `open()`.
(4) If there is not a file with the redirected path, perform `open()` using the original path.

Files served by McKernel are deleted in the following two ways. The first one is McKernel deletes files for threads when destroying them. The second one is `mcctrl` deletes files created at the boot time of McKernel.

File accesses to files served by McKernel are performed using the following steps.

(1) Linux detects access to a file served by McKernel and calls the call-back function registered by `mcctrl` using `file_operations` interface
(2) `mcctrl` redirects the access request to McKernel using IKC
(3) McKernel writes expected value to memory owned by Linux
(4) McKernel notifies `mcctrl` of the completion of the memory write using IKC
(5) `mcctrl` sends the result back to Linux using `file_operations` interface

4. Related Work

This section surveys related work covering studies on operating system design for many-core CPUs, lightweight kernels for high-performance computing, and existing hybrid kernel solutions.

4.1 Kernels for Multi/Many-cores

Operating system organization for manycore systems has been actively researched in the past decade. In particular, issues related to scalability over multiple cores have been considered.

K42 [10] was a research OS designed from the ground up to be scalable. Similarly how IHK/McKernel selectively implement a set of performance sensitive system calls on the LWK side, K42 allowed the application to circumvent the Linux APIs and call native K42 interfaces. However, it involved a significant entanglement with Linux which made it difficult to keep track of the latest kernel changes. Although McKernel also relies on Linux, as discussed above, one of its important design criteria is to minimize the engineering effort required to keep it up-to-date with the rapidly evolving Linux kernel codebase.

Corey [11], an OS designed for multicore CPUs, argues that applications must control sharing in order to achieve good scalability. Corey proposes several operating system abstractions that allow applications to control inter-core sharing. The IHK/McKernel infrastructure enables fine grained control over sharing by allowing explicit resource partitioning and the execution of multiple LWK instances.

Turning towards multiple kernels, Tessellation [12] and Multikernel [13] are built upon the observation that modern node hardware resembles a networked system and so the OS should be modelled as a distributed system as well. The Tessellation project [12] follows a resource partitioning approach called Space-Time Partitions. It divides CPU cores into groups called cells, where each cell is responsible for a particular application or some system services. Since applications and system services can be assigned to separate cells, To some extent, Tessellation's structure also resembles IHK/McKernel, where HPC workloads are explicitly assigned to LWK cores while system daemons reside on the Linux partition. On the other hand, Multikernel [13] runs a small kernel on each CPU core and the OS is built as a set of cooperating processes, where each process is running on one of the kernels and communicating via message passing. Similarly to Multikernel, the IHK/McKernel model employs a low level message passing facility which enables communication between the two types of kernels.

4.2 Lightweight Kernels

Lightweight kernels developed from scratch and designed for high performance workloads have been around for over two decades now. Notably, Catamount [14] from Sandia National Laboratories was one of the first systems which has been successfully deployed on a large scale supercomputer. IBM's BlueGene line of supercomputers have also been running an HPC targeted lightweight kernel called the compute node kernel (CNK) [15]. The most recent in Sandia National Lab's lightweight compute node kernels line of effort is Kitten [16], which distinguishes itself from their prior LWKs by providing a more complete Linux-compatible user environment. As opposed to McKernel, Kitten relies on the Linux bootstrapping code, it directly takes advantage of the Linux device drivers, but it implements most of the HPC sensitive kernel services independently. It also features a virtual machine monitor via Palacios [17] that allows full-featured guest OSs to be deployed along the lightweight kernel. Contrary to McKernel, Kitten requires a set of patches to be carried over to newer Linux kernel versions.

Another approach to lightweight kernels is to start with Linux, but apply heavy modifications to meet HPC requirements ensuring low noise, scalability and predictable application performance. Cray's Extreme Scale Linux [18] and ZeptoOS [19] follow this approach. Techniques, such as eliminating daemon processes, simplifying the scheduler or replacing the memory management system are often applied. There are primarily two problems with this approach. First, the heavy modifications occasionally break Linux compatibility, which is not desirable. Second, because HPC tends to follow (or rather dictate) rapid hardware changes that need to be reflected in kernel code, Linux often falls behind with the necessary updates which results in difficulties for maintaining Linux patches. In contrast, IHK/McKernel is trying to embrace the bests of both worlds aiming at full Linux compatibility without sacrificing LWK performance.

4.3 Hybrid Kernels for HPC

FusedOS [20] was the first proposal to combine Linux with an LWK. It's primary objective was addressing core heterogeneity between system and application cores and at the same time providing a standard operating environment. Contrary to McKernel, FusedOS runs the LWK at user level. In the FusedOS prototype, the kernel code on the application core is simply a stub that offloads system calls to a corresponding user-level proxy process called CL. The proxy process itself is similar to that in IHK/McKernel, but in FusedOS the entire LWK is implemented within the CL process on Linux. The FusedOS work was the first to demonstrate that Linux noise can be isolated to the Linux cores and avoid interference with the HPC application running on the LWK cores. This property has been one of the main driver for the IHK/McKernel model.

Argo [21] is one of the DOE OS/R project targeted at applications with complex workflows. They envision using OS and runtime specialization inside the compute node relying on containers. In Argo's architecture, each node may contain a heterogeneous set of compute resources, a hierarchy of memory types with different performance (bandwidth, latency) and power characteristics. Given such a node architecture, Argo expects to use a ServiceOS like Linux to boot the node and run management services. It then expects to run different ComputeOS containers that cater to the specific needs of the application.

Hobbes [22] is another DOE founded Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to explicitly support application composition, which is emerging as a key approach for applications to address scalability and power concerns anticipated with coming extreme-scale architectures. Hobbes makes use of virtualization technologies to provide the flexibility to support requirements of application components for different node-level operating systems and runtimes. At the bottom of the software stack, Hobbes relies on Kitten [16] as its LWK component, on top of which Palacios [17] is in charge to serve as a virtual machine monitor. Contrary to Hobbes, IHK/McKernel currently does not consider utilizing virtualization technology.

Finally, the project which resembles IHK/McKernel the most is Intel's mOS [23]. mOS also explicitly partitions CPU cores and physical memory and assigns part of the resources to Linux, while the rest is utilized by an HPC LWK. Although mOS and McKernel has very similar goals, the main difference is in their system call offloading mechanism. Unlike relying on a low level messaging layer such as IKC in IHK/McKernel's, mOS' LWK retains Linux compatibility at the kernel data structures level and migrates threads to the Linux partition when system call offloading if performed.

5. Conclusion

Operating system design for next-generation supercomputer faces challenges which are produced by the changes in the environment, i.e., many-core will be used and the Linux-compatible environment should be provided. We have been investigating and developing a design of a hybrid Linux plus LWK approach, called McKernel, to deal with those challenges, and detailed the approaches in this paper. In summary, the compact nature of LWK makes it easy to exploit change in memory hierarchy and facilitates creating application specific kernels, and LWK plus Linux hybrid model makes it easy to provide Linux compatible environment and reducing the cost of tracking Linux kernel changes and delegating OS services is effective in reducing cache pollution and OS noises.

The progress of McKernel development is reported briefly.

The implementation of system calls is validated using Linux Test Project (LTP) test programs and gdb test suite. System call part of LTP which has 1013 tests are used and all but one tests passes. “ gdb.base ” part of gdb tests which has 312 tests are used and 216 tests pass so far.

6. Acknowledgements

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

References

[1] Soma, Y., Gerofi, B. and Ishikawa, Y.: Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach, *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, New York, NY, USA, ACM (2014).

[2] Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A. and Ishikawa, Y.: Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures, *High Performance Computing (HiPC), 2014 21th International Conference on*, HiPC '14 (2014).

[3] Reinders, J.: Knights Corner: Your Path to Knights Landing, `https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf` (2014).

[4] The TOP500 project: Supter Computer TOP500, `http://www.top500.org`.

[5] Shimosawa, T.: Operating System Organization for Manycore Systems, PhD Thesis, Computer Science Department, University of Tokyo (2012).

[6] Hirai, K., Otawa, T., Okamoto, T., Ninomiya, A., Sumimoto, S., Takagi, M., Gerofi, B., Yamaguchi, N., Ogura, T., Kameyama, T., Hori, A. and Ishikawa, Y.: Issues of Batch Job Execution Environment using Many-core OS for HPC, *IPSJ SIG Notes 2015-OS-133* (2015).

[7] Duell, J.: The design and implementation of Berkeley Lab Linux Checkpoint/restart, Technical report, Lawrence Berkeley National Laboratory (2000).

[8] Gerofi, B., Shimada, A., Hori, A. and Ishikawa, Y.: Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures, *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (2013).

[9] Gerofi, B., Shimada, A., Hori, A., Masamichi, T. and Ishikawa, Y.: CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores, *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, New York, NY, USA, ACM, pp. 73–84 (2014).

[10] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A. and Uhlig, V.: K42: Building a Complete Operating System, *SIGOPS Oper. Syst. Rev.*, Vol. 40, No. 4, pp. 133–145 (online), DOI: 10.1145/1218063.1217949 (2006).

[11] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y. and Zhang, Z.: Corey: an operating system for many cores, *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pp. 43–57 (2008).

[12] Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K. and Kubiatowicz, J.: Tessellation: Space-time Partitioning in a Manycore Client OS, *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, Berkeley, CA, USA, USENIX Association, pp. 10–10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1855591.1855601⟩ (2009).

[13] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, ACM, pp. 29–44 (2009).

[14] Kelly, S. M. and Brightwell, R.: Software architecture of the light weight kernel, Catamount, *In Cray User Group*, pp. 16–19 (2005).

[15] Giampapa, M., Gooding, T., Inglett, T. and Wisniewski, R. W.: Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, IEEE Computer Society, pp. 1–10 (online), DOI: 10.1109/SC.2010.22 (2010).

[16] Sandia National Laboratories: Kitten: A Lightweight Operating System for Ultrascale Supercomputers (Accessed: Jan, 2015), `https://software.sandia.gov/trac/kitten`.

[17] Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M. and Brightwell, R.: Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing, *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12 (online), DOI: 10.1109/IPDPS.2010.5470482 (2010).

[18] Cray Inc.: Cray Linux Environment (CLE) 4.0 Software Release Overview (Accessed: Jan, 2015), `http://docs.cray.com/books/S-2425-40/S-2425-40.pdf`.

[19] Yoshii, K., Iskra, K., Naik, H., Beckmanm, P. and Broekema, P. C.: Characterizing the Performance of Big Memory on Blue Gene Linux, *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, IEEE Computer Society, pp. 65–72 (2009).

[20] Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D. and Wisniewski, R.: FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment, *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pp. 211–218 (online), DOI: 10.1109/SBAC-PAD.2012.14 (2012).

[21] Argonne National Laboratory: Argo: An Exascale Operating System (Accessed: Jan, 2015), `http://www.mcs.anl.gov/project/argo-exascale-operating-system`.

[22] Brightwell, R., Oldfield, R., Maccabe, A. B. and Bernholdt, D. E.: Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R, *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, New York, NY, USA, ACM, pp. 2:1–2:8 (online), DOI: 10.1145/2491661.2481427 (2013).

[23] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R. and Riesen, R.: mOS: An Architecture for Extreme-scale Operating Systems, *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, New York, NY, USA, ACM, pp. 2:1–2:8 (online), DOI: 10.1145/2612262.2612263 (2014).