Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach

Yuki Soma, Balazs Gerofi, Yutaka Ishikawa
Graduate School of Information Science and Technology,
The University of Tokyo, JAPAN
somay@il.is.s.u-tokyo.ac.jp, bgerofi@il.is.s.u-tokyo.ac.jp, ishikawa@is.s.u-tokyo.ac.jp

ABSTRACT

Page-based memory management (paging) is utilized by most of the current operating systems (OSs) due to its rich features such as prevention of memory fragmentation and fine-grained access control. Page-based virtual memory, however, stores virtual to physical mappings in page tables that also reside in main memory. Because translating virtual to physical addresses requires walking the page tables, which in turn implies additional memory accesses, modern CPUs employ translation lookaside buffers (TLBs) to cache the mappings. Nevertheless, TLBs are limited in size and applications that consume a large amount of memory and exhibit little or no locality in their memory access patterns, such as graph algorithms, suffer from the high overhead of TLB misses.

This paper proposes a new hybrid kernel design targeting many-core CPUs, which manages the application's memory space by segmentation and offloads kernel services to dedicated CPU cores where paging is utilized. The method enables applications to run on top of the low-cost segmented memory management while allows the kernel to use the rich features of paging. We present the design and implementation of our kernel and demonstrate that segmentation can provide superior performance compared to both regular and large page based virtual memory. For example, running Graph500 on top of our segmentation design over Intel's Xeon Phi chip can yield up to 81% and 9% improvement compared to utilizing 4kB and 2MB pages in MPSS Linux, respectively.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Storage Management—*Virtual Memory*

Keywords

Segmentation; Hybrid kernel; Manycore; Xeon Phi

ROSS '14, June 10, 2014, Munich, Germany

http://dx.doi.org/10.1145/2612262.2612264

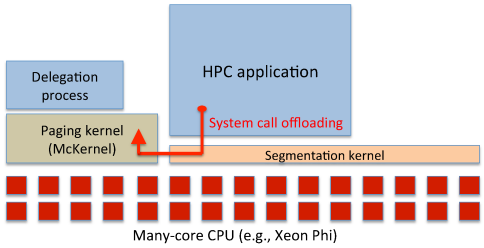

Figure 1: Hybrid paging and segmentation kernel architecture. ***Certain system calls issued by the application running over the thin segmentation kernel are offloaded to the paging kernel.***

1. INTRODUCTION

Most of the current OSs use page-based memory management (paging) to provide virtual memory with rich features such as memory fragmentation avoidance and fine-grained protection control. Paging enables these features by defining the virtual to physical address mapping in steps of small memory units, called pages. This mapping (i.e., the page table) is transparently referenced by the memory management unit (MMU) during address translation, each time the execution involves a memory access. Because page tables also reside in the physical memory, modern CPUs employ translation lookaside buffers (TLBs), which cache page table entries in order to avoid the cost of memory accesses each time a lookup in the page table is performed.

Supporting TLB is not, however, a sufficient solution in all situation. Since TLBs are caches, their effectiveness inherently depends on the application's memory footprint, as well as the memory access pattern's spatial and temporal locality. Recent research suggests that TLB cannot eliminate the paging cost completely and leaves an overhead of 5-14% of the execution time even for usual applications [10]. The cost can presumably grow as high as 50% of the execution time for applications which have *unfortunate* access patterns [8, 18].

Taking the emergence of data-centric workloads and large-scale data analyses in enterprise and scientific computing into account, the effectiveness of TLBs will likely further decrease in the future [24]. The continuing increase of the amount and variety of digital data leads to more diverse and complex data processing so that meaning can be extracted from the data. At the same time, more and more data are kept in memory so that response latency of online

services can be minimized. These processing methods have complicated and close to random access patterns, which are unsuitable for TLBs. A typical example is Facebook, where most of the complex friend linkage data are stored in memory, which renders the memory footprint large and the access pattern having little or almost no locality [21].

On the other hand, the latest trend of CPU architecture is integrating a large number of simple CPU cores (i.e., manycore CPUs), because improving single core performance by extracting instruction level parallelism makes the hardware complex, results in unreasonable increase of die area, design and verification time, and power consumption [20, 12]. This architectural trend has created another source of cost for paging based virtual memory with TLBs: the consistency problem. To keep each CPU core's TLB consistent after memory allocation and deallocation, TLBs must be invalidated by software, which if often referred to as *TLB shootdown*. Reportedly, this cost can grow over 10% when dozens of CPU cores are utilized by applications that frequently update memory mappings and page permissions [25]. Reducing the cost of paging will thus become an important issue in the future. TLBs also consume non-negligible energy since they are accessed on every memory or cache access. Industrial report shows that 3-14% of power consumption of a core (including cache) can be accounted to TLBs [27].

One way to avoid these taxes of paging is not using paging: using segmentation. This approach can eliminate all problems related to paging. Also, since the size of segments can be variable, there is no need for re-designing the hardware in response to the increase in memory size, unlike paging, where changing page size necessitates both hardware and software changes. Moreover, without paging, the design of CPU cores becomes simpler by eliminating hardware page walk handlers, TLBs and other related mechanisms. To use segmentation, however, one must consider how to supplement the lack of features that paging offers, such as avoidance of memory fragmentation and per-page access control, which functions may be necessary to some applications or OS kernels.

We introduce a new design of OS kernel in which the main OS kernel runs on paging, but certain applications (those targeting high performance) can be executed using segmentation based memory model. In this design, the OS kernel using paging (paging kernel) works in harmony with another tiny kernel, running on another CPU core, which enables the segmented memory (segmentation kernel). High level overview is shown in Figure 1. The segmentation kernel provides a minimalistic execution environment for applications and delegates kernel functions to the paging kernel. Utilizing manycore architectures, we use a few cores to execute the paging kernel and the rest of the cores may run over segmentation. Such luxurious usage of CPU cores is not available in single- or multi-processor architectures since applications do time sharing over the cores. We describe our design, reveal some details of the implementation and provide quantitative measurements on the performance benefits of segmentation over paging.

The rest of this paper is organized as follows. Section 2 provides background and gives on overview of segmentation, Section 3 discusses the design and implementation of our hybrid kernel. Section 4 provides experimental results, Section 5 surveys related work, and finally, Section 6 concludes the paper.

2. BACKGROUND

2.1 Segmentation in x86 Architecture

The x86 (or x86-64) architecture has several modes of operation. In this section we describe segmentation on protected mode, where segmentation can be used without any restrictions while the length of virtual address and registers are limited to 32 bits. In segmentation memory model on protected mode, the virtual address space is defined as a set of variable-sized address spaces named *segments*. Each of these has its own start (base) physical address, size, and other attributes. These parameters of segments are set to be matched with their own intended uses.

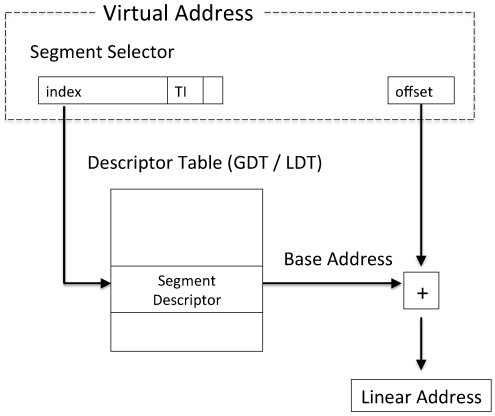

Figure 2: Address Translation using Segmentation.

The mechanism of segmentation is depicted by Figure 2. A virtual address in segmentation is a pair of a *segment selector*, which selects a segment to use, and an offset to the base address of the segment. There are 6 segment registers on one core, and they are named CS (Code Segment), DS (Data Segment), ES, FS, GS and SS (Stack Segment), respectively. CS is used during instruction fetch, SS is used for instructions which access the stack, such as push/pop and move instruction which uses the EBP or ESP register as the offset of the address. DS is used for regular memory access. The other segment registers, FS, GS are used to make spaces to store thread-local or core-local data.

Specifically, the segment selector selects an entry in *segment descriptor tables*, which is called *segment descriptor* and corresponds to a segment. A segment descriptor is a length of 64 bits, and holds the base address, size, access permission and other attributes of the segment. A segment descriptor table is an array of the entry. There are two types of segment descriptor tables: the *GDT* (Global Descriptor Table) and the *LDT* (Local Descriptor Table). The primary difference between GDT and LDT is the way how the address of the table is specified. Although LDT is useful when several processes run on the same core because LDTR changes automatically at hardware task switch, we use only GDT in our implementation, because we currently assume that only a single thread runs on a core.

To translate a virtual address to physical address, the selector part of the virtual address specifies a segment descriptor. A segment selector is a length of 16 bits, which holds a

bit (TI in Figure 2) to choose between LDT or GDT and a index in the GDT or LDT. After a entry is selected, the size and other attributes of the segment are checked to verify the validity of the memory access, and if valid the physical address is calculated by adding offset to the base address of the segment. To accelerate the translation, segment descriptors are cached in segment registers and LDTR when a segment selector is loaded on these registers, which eliminates any additional memory accesses from the translation.

The x86 architecture offers a memory management mechanism which uses paging and segmentation at the same time. In this mechanism, virtual addresses are translated to linear address by segmentation and then to physical address by paging. What we call physical addresses in the above description of segmentation is strictly linear addresses. In this research, we enable only segmentation and clear the paging cost.

In 64 bit mode, where virtual address and some registers are a length of 64 bits, to the best of our knowledge it is impossible to disabling paging [16]. In addition, in segmentation with paging in 64 bit mode, the base addresses of CS, DS, ES, SS are fixed at zero and attributes of segment descriptors are almost ignored. Paging is thus the primary memory management mechanism in current x86-64 architectures.

2.2 McKernel

In this section we give a short description of *McKernel*, a light-weight kernel for manycore architectures being developed primarily by the University of Tokyo and RIKEN AICS [26, 14]. Although our proposed design is implemented on top of McKernel, we believe the design can be applied to other OS kernels on manycore architectures. Design principles of McKernel are small memory and cache footprint and scalable kernel data structure. To achieve these goals, McKernel supports only necessary functions of OS kernel such as memory and process/thread management and handling system calls, and some unconventional data structures of OS kernels [14]. It can be used easily and is suitable for applications with high parallelism, because it maintains Linux ABI and supports facilities for parallel computing such as pthreads and OpenMP.

Mckernel runs on Intel's Xeon Phi coprocessor, which is currently non-bootable PCI Express device and thus it works with the help of Linux running on the multi-core host. Both of these two kernels utilize our Interface for Heterogeneous Kernels (IHK), which enables memory mapping, interrupt sending and DMA between a host and a coprocessor. IHK on manycore is implemented as a kernel code library, and IHK on host is implemented as a kernel module and has duties of initializing coprocessor and booting, shutdown and other management of a OS on coprocessor. Two kernels communicate with Inter Kernel Communication (IKC), which is implemented using IHK and provides asynchronous messaging facilities.

To execute an application on the co-processor, we use a special program, *mcexec*, on host Linux. This program takes a statically-linked application as the parameter and depute this to the manycore. The process image of the application is loaded into the memory of the coprocessor and then McKernel starts execution the application. The process image is memory-mapped to that of mcexec waiting for offloading of system calls from McKernel through IKC. Mckernel handles

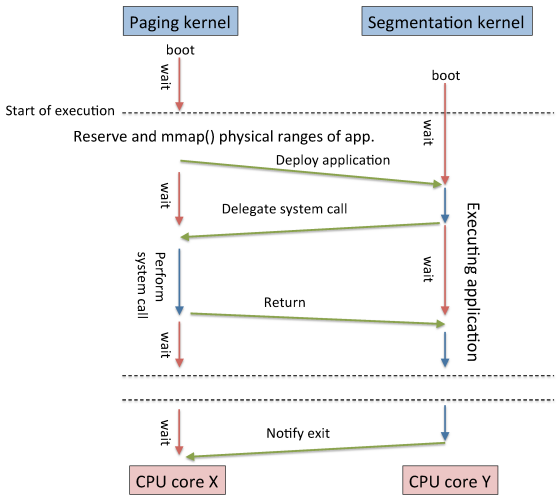

Figure 3: Flow chart of execution from bootstrap to exit.

simple system calls by itself, but complex system calls such as file I/O is forwarded to mcexec and processed in host Linux. Note that this offloading mechanism is independent from the one between the segmentation kernel and McKernel itself.

In current design of McKernel, only one application can be executed on McKernel at the same time. Implementation of the proposed design described later is thus also restricted to that execution model.

3. DESIGN AND IMPLEMENTATION

3.1 General Design

In our design, two types of kernels are working simultaneously on the manycore unit. One is a kernel which uses paging memory management model (paging kernel), and the other is a tiny kernel which uses segmentation (segmentation kernel). CPU cores can be divided into these two groups and by default we boot a paging kernel on the first core. User communicates only with the paging kernel and she can request execution of applications on top of the segmentation kernel(s). The segmentation kernel executes an application and when system calls or other kernel requests are issued, it delegates the requests to the paging kernel. Of course, applications may be executed on the paging cores as well.

Figure 3 presents a flow chart describing the interaction between the two kernels from the bootstrap to the end of an application when executed on the segmentation kernel. Unlike in paging, in segmentation large enough contiguous physical address ranges are required for both kernel and user address spaces. The segmentation kernel ranges are hence reserved by the paging kernel during bootstrap, and user ranges are either reserved during boot or right before the execution of an application. The user ranges are also memory-mapped to a virtual address space of the paging kernel so that it can read and write to/from the user space when handling delegated system calls.

To execute an application on the segmentation cores, a program which delegates the application to the segmentation kernel is executed on the paging kernel. The delegation in the program is performed by a special system call, which

writes the process image of the application to the reserved user range. Subsequently, it notifies the segmentation kernel to start the execution, waits for system call requests from the segmentation kernel until the end of the application's execution and receives notification when the execution is finished.

When a system call is issued, the segmentation kernel first decides based on the system call number whether it delegates the call to the paging kernel or handles by itself. If the system call is delegated, the paging kernel handles the system call and sends the return value to the segmentation kernel. The segmentation core simply waits for the return value in this case. When the application finished execution, the final return value is sent to the paging kernel, and the segmentation core starts waiting for the next application. An error code is sent as a substitute of the final return value if the application finishes unexpectedly.

If a return value or an error code is sent from the segmentation kernel, the paging kernel then returns from the delegation system call and the delegation program exits.

3.2 Discussion on the Design

Our proposed design has three advantages. First, applications can run on top of the low-cost segmentation memory management model, which can boost performance and can eliminate paging related power consumption, such as TLBs and hardware page table walker. This merit will become more important in the future since the detrimental effect of paging on performance will likely increase as described earlier.

Second, OSs can still use the functions of paging if required. Since segmentation offers limited, coarse-grained features compared to paging, an OS could become less reliable and/or secure in case segmentation was used exclusively as the method for memory virtualization. In this design, CPU cores running the paging kernel can still utilize the fine-grained features of page based virtual memory.

Third, it costs less to implement this design than to implement a full OS from scratch, because the segmentation kernel can take advantage of various functionalities provided by the paging kernel. For instance, during the implementation of the proposed design, we could directly re-use source codes of most delegated system calls to the paging kernel (with little or no modification), thus eliminating the need for extra development. The simple mechanism of segmentation kernel also makes the kernel code significantly smaller.

An important design consideration is the decision which system calls are delegated to the paging kernel. Although it is impossible for some system calls to be implemented without the help of paging, most system calls could be either delegated or handled directly. There are mainly three points to consider when deciding whether a system call is handled by the segmentation kernel or the paging kernel. First, the call frequency of a specific system call. Since delegation of system calls inherently takes some time due to the communication between the two kernels, if system calls are issued frequently the accumulated cost of delegation may increase execution time. It is hence a good decision to implement frequently called system calls in the segmentation kernel to minimize delegation overhead. Second issue is whether a system call must be handled in the thread which issues the system call. There are system calls which are related to thread local variables or inter-thread communication such as synchronization. These system calls must not be delegated to the paging kernel. Third point is whether or not there is any useful code in the paging kernel that could implement a system call. Due to the delegation cost, if the cost to implement a system call in the segmentation kernel is lower than that in the paging kernel, implementation in the segmentation kernel might be the better choice.

Another consideration is when to reserve and memory-map user ranges for segmentation: during bootstrap or directly before execution. Both of these have pros and cons. If choosing the latter, we can make more flexible decisions on the size of the allocated memory based on the estimation of the memory footprint of the application, leading to avoiding reserving unused memory space and leaving more memory to applications executed on top of the paging kernels. There is also a possibility that physical memory space is fragmented at the start of execution and that the application cannot get enough contiguous memory area to be executed. Although memory compaction can solve this problem at most case, it imposes more overhead at the start of execution. If allocating takes place during bootstrap, fragmentation is not a problem. In contrast, this choice may reserve surplus unnecessary memory space and cause termination of an application on the paging cores due to shortage of physical memory.

There are also several limitations in our design. To benefit the most, it is important to choose the suitable applications. If the application inherently requires features of paging like per-page access control, it cannot be run on a segmentation kernel. Also, applications which have complex allocation and deallocation patterns are unsuitable because they are subject to cause memory fragmentation. Applications with high parallelism are more appropriate because they don't waste segmentation cores. If running multiple applications on segmentation cores, the memory footprint of each applications must be known beforehand to provide sufficient memory.

3.3 Implementation

The proposed design is implemented on Intel's Xeon Phi coprocessor, where McKernel is used as the paging kernel. The Xeon Phi coprocessor provides hyper-threading mechanism and four *hyper-threads* can be executed on a physical processor core. From this point on, we refer to "a core" as a hyper-thread.

There are some restrictions of the current prototype implementation. First, a segmentation kernel must run in protected (32-bit) mode since segmentation without paging is (to the best of our knowledge) not available on 64-bit mode in current x86 architectures. Nevertheless, McKernel does run 64-bit kernel code. This constraint renders an application's memory usage smaller than 4 GB. Also there are several interface problems when communicating between McKernel and a segmentation kernel, such as the differences of system call numbers.

The second is that stack, data and code segments can't be separated in a physical address space because they must have the same base address due to linkers and compilers. Only two contiguous physical address spaces thus need to be reserved as a kernel data/stack/code segment and a user segment, as shown in Figure 4. The kernel space is reserved when the process image of the segmentation kernel is loaded into an area of physical memory at the time of loading McK-

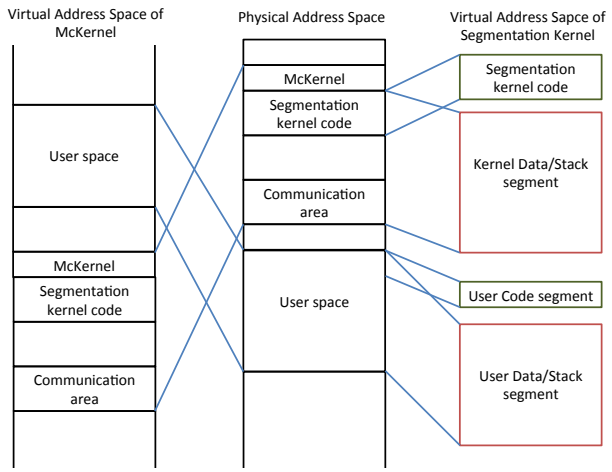

Figure 4: Memory mappings of an application executed by the segmentation kernel.

ernel's process image, and the user space is reserved and memory-mapped later. Although we could eliminate this restriction by modifying the linker and the compiler, we opted to go for the simpler solution.

The design gives almost the same interface to applications as the paging kernel, hence applications for this design are written and compiled in the same way as for McKernel, except the modification of the linker script to set the start address of the text section to 0.

The two kernel communicates on a memory space named the *communication area*, which is reserved and memory-mapped to the virtual address space of McKernel as well. The area is used when delegating system calls to McKernel and handling the special system calls described below.

Two system calls were added to McKernel for users to access segmentation cores: init_core , which boots segmentation cores, and load_core, which is called just after a issue of init_core and starts to execute an application on segmentation cores.

The init_core system call first reserves the user space of segmentation kernel and memory-maps that regions to an area of the virtual address space of McKernel. User space is reserved just before the execution on a segmentation kernel because it must be memory-mapped to an area in the user space of the delegation program due to the system call delegation to the host Linux. McKernel then boots all segmentation cores, waits until all cores are booted and returns from the system call. Each segmentation core gets a boot request with some variables such as the base physical address of the kernel code, which is needed to set the GDT. The bootstrap includes the setting of the GDT and other usual settings. A segmentation core notifies McKernel when booted, and spinlocks with a exec_flag of the core in the communication area.

The load_core system call gets the start address of a memory-mapped ELF object image as the argument, and first expands it to a process image of the application. Tload_core then selects a segmentation core to execute the application, and sends the selected core the entry point address, the initial value of stack pointer and the size of data segment through the communication area. When all values are written, exec_flag is set so that the segmentation core can start to execute the application. McKernel then waits for the termination of the execution and system call delegation requests. The unlocked segmentation core sets the GDT (user code/data/stack segment), switches to user mode and execute the application.

If a system call is delegated during execution, arguments of the system call are passed through a special structure in the communication area. At delegation, a segmentation core locks the structure to prohibit others cores to overwrite the argument which is used by McKernel. Segmentation cores notify the arrival of a system call to McKernel by setting a flag of system call after all arguments had been written to the communication area. The segmentation core then waits until the arrival of a return value of the system call with which it returns to user mode.

4. EVALUATION

4.1 Experimental Setup

All experiments were carried out on Intel's Xeon Phi coprocessor, with hardware specifications shown in Table 1. Data and instruction caches as well as TLBs are per-core resources.

Processor	Intel's Xeon Phi Knight Corner 60 cores, 4 hyper-threads/core 1.0 GHz	
L1 DTLB	4KB: 64-entry 64KB: 32-entry 2MB: 8-entry	4-way assoc.
L1 ITLB	4KB: 32-entry	
L2 TLB	4KB&64KB&2MB: 64-entry	
L1 D cache	32KB	8-way assoc.
L1 I cache	32KB	line size:
L2 D/I cache	512KB	64 byte
RAM (GDDR5)	8GB	

Table 1: Hardware specification. ***D and I stands for data and instruction, respectively.***

Note that each core provides 64 L2 TLB entries, regardless the size of the mapping. Consequently, using large pages the TLB cache can cover 128MBs of virtual memory, which as we will see later corresponds to our measurements.

4.2 Results

4.2.1 Random Memory Access

The first measurement we performed is a micro benchmark to assess random memory access performance for which we used a standard code from the HPC Challenge Benchmark [2]. We used array sizes up to 2GB for this experiment and rewrote the algorithm in x86 assembly to make sure that the exact same codes run on both of 32-bit kernel (our design) and the 64-bit kernel (McKernel). Random access is an important workload in our case because its access pattern exhibits close to zero locality and thus it provides insights to what extent segmentation may improve performance.

Results are shown in Figure 5 as the function of the amount of memory used. As seen, random access with segmentation is significantly faster than both of 4kB and 2M pages. If more memory could be used, the difference would be likely

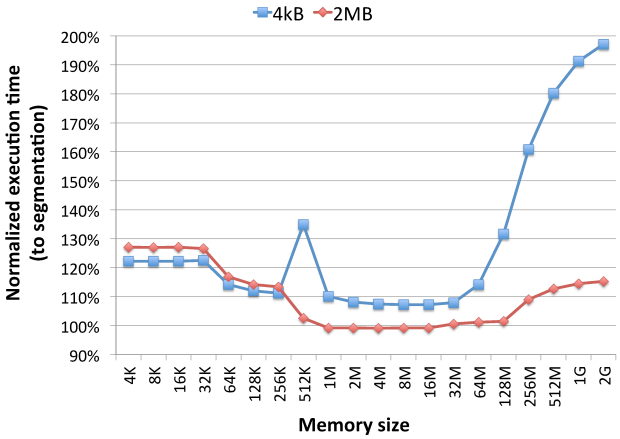

Figure 5: Normalized execution time of random access benchmark using 4kB/2MB pages compared to segmentation as the function of memory footprint.

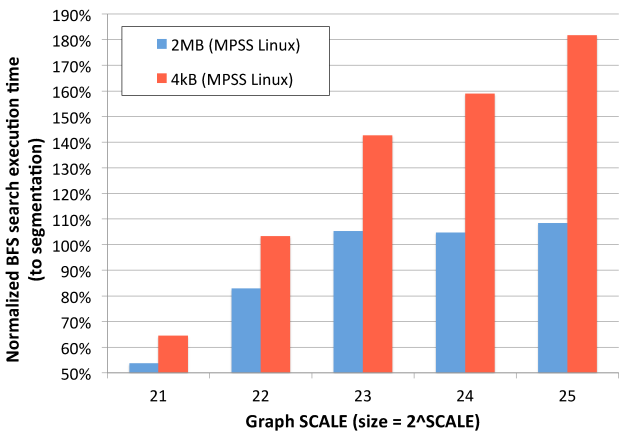

Figure 6: Normalized average execution time of BFS search in Graph500 using 4kB/2MB pages on MPSS Linux compared to segmentation with McKernel as the function of graph scale.

even larger. In fact, approximations provided by [8] assess that 83% and 53% of the execution time would be spent on page walks accessing a 64GB array if using 4KB pages and 2MB pages, respectively. Our actual measurements reveal similar tendencies. At memory usage of less than 32kB, segmentation is 22% faster than regular pages and 26% faster than large pages. This is because TLB lookup cost of cache access is eliminated on segmentation. At 64kB a non-linear increase of execution time happens and the ratio between segmentation and paging becomes 11-17% due to the overflow of the L1 data cache. The gap between segmentation and 4kB pages is especially larger at 512kB, because at this point it exceeds the size of L2 TLB, but not the L2 cache, and thus resulting in relatively higher impact of the L2 TLB misses. From 32MB on, performance gap between segmentation and regular pages increases steadily and reaches almost 200% at 2GB.

Although segmentation is only about 15% faster than large pages at 2GB, the memory footprint is relatively smaller compared to memory size of current computer systems, and thus, the performance gap may increase significantly in large memory practical use cases, assuming segmentation would be available in 64-bit execution mode.

4.2.2 *Graph500*

In order to explore the benefits of segmentation on a real application we chose to evaluate our design using the Graph500 benchmark [1]. Graph500 is a community effort backed by over 50 international HPC experts to create a set of benchmarks that truly represents data intensive applications in areas such as cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks. The benchmark generates a undirected graph and consequently performs multiple breadth-first search operations on it.

We measured the performance of Graph500 in three settings. First, running it over our segmentation design where system calls are offloaded to McKernel cores, as described earlier. We then compared the results to executions over Intel's MPSS Linux (the default software stack provided for the Xeon Phi) using 4kB and 2MB pages. In order to make the comparison fair, we use the same random seed for graph generation and the same set of start vertices for the BFS search. Also, we utilize graphs with large number of vertices and low edge factor so that we can get relatively large graphs even in the 32 bit address space (i.e., for segmentation). Note, that MPSS Linux runs the benchmark in 64-bit mode. In our prototype implementation, we currently only support one thread and thus we provide results only for sequential execution. Support for multi-threaded execution is part of our future agenda.

Results are shown in Figure 6. The X axis represents the size of the graph up to the maximum scale we could squeeze into the 32 bit address space. Y axis is the execution time of the average BFS search when using 4kB and 2MB pages normalized to the segmentation performance.

As seen, for small graph size (SCALE equals 21), both 4kB and 2MB pages perform better than segmentation. We believe this might have something to do with the fact that the Xeon Phi doesn't cache all segment registers and accessing memory that is entirely covered by the TLB entries may be slightly faster than using segmentation or perhaps some differences between the 32-bit and 64-bit executions. Nevertheless, at SCALE of 23 performance of both 4kB and 2MB pages starts to degrade when compared to segmentation. As seen, for SCALE 25, the benefits of segmentation becomes approximately 81% and 9% against 4kB and 2MB page sizes, respectively. We believe these results are extremely promising, considering that increasing the memory size we would very likely see even more benefits.

5. RELATED WORK

To the best of our knowledge, our design is the first attempt to utilize segmentation mechanism (with an actual implementation) for the purpose of high performance computing. The most relevant research similar to our approach is *direct segment* [8], a work that appeared in parallel with our efforts. The authors of direct segment showed that today's big-memory workloads seldom exploit the functions of paging and proposed a hardware extension, called *direct segment*, which translates a range of virtual addresses to physical address without any memory access while paging works

normally in the other parts of virtual address space. The OS reserves a contiguous physical address range, maps it into the virtual address space via a so called *primary region*, and uses this space when memory spaces with read/write access permission is requested. This enables the application to manage most of its memory by segmentation in the low-cost part and can reduce TLB misses of big-memory workloads significantly.

There are three main points which distinguish our work from this effort. First, the most important limitation of direct segment is that it explicitly requires hardware modification. Although some features of segmentation in the current 64-bit x86 architecture are disabled by default, which restricted us to implement our design only in 32-bit mode, we believe *re-enabling* full support for segmentation in 64-bit mode would be relatively straightforward. Second, due to the fact that there is no hardware available with direct segment support, authors in [8] could only approximate the benefits of such system by carefully calculating the cost of TLB misses. In contrast, we provide an implementation and reveal actual measurements on real hardware. Finally, although direct segment also combines segmentation and paging, the authors intention is to utilize paging and segmentation in tandem on the same CPU core which in fact is the reason that makes hardware modifications necessary. On the contrary, we utilize a hybrid approach with CPU core specialization, a technique increasingly viable on many-core architectures, to offload system services to CPU cores where rich features of paging can be taken advantage of.

Several studies on paging overhead proposed more efficient TLB designs and they often employ naturally-occurred contiguity of address mappings. SpecTLB [7] infers translated addresses from contiguity of mapping at TLB misses and speculatively continues execution in parallel with page walks. CoLT [23] is another design which stores multiple consecutive address mappings in single TLB entries. MMU caches along with the necessary software support were proposed in [11], where the intention is to coalesce consecutive PTEs in MMU caches and to share MMU caches by multiple CPU cores. Most of these works, however, only *mitigate* paging cost, and cannot eliminate it completely, although they retain all features of paging. Also, most of these studies make hardware even more complicated. In our design, paging cost is eliminated exhaustively, but hardware design remains the same or could become even simpler.

Support for large pages are one of the current solutions to high overhead of paging. If modifying applications and using libraries such as libHugeTLBFS [4] or using the mmap system call with a special flag, large pages can be enforced explicitly. Mechanisms to automatically use large pages were also studied [28, 19] and recently introduced to Linux known as Transparent Hugepage [6]. Architectures, which support very large mappings, can also provide an alternative solution to segmentation via employing very large TLBs, as demonstrated by Yoshii et al. for BlueGene/P [29]. While large pages currently succeed in considerably mitigating paging overhead, their success will not continue forever due to various difficulties in applying them to ever larger memory sizes as Basu et al. argue in [8].

Some extinct OSs used segmentation, but both the details and usage of these mechanisms are very different from that of today's x86 architecture. For example, Multics is a famous old OS used segmentation [9]. However, Multics used segmentation on top of paging, which is the opposite of our design. iMAX [17] is a OS on iAPX 432 microprocessor architecture, which doesn't support paging but only segmentation [15]. Although the mechanism of segmentation in iAPX 432 is similar to that of x86, check of access right is far more detailed in iAPX 432. iAPX 432's purpose of segmentation is not performance but protection of data abstraction and due to its complex design, the machine was significantly slower than other computer systems at that time, which resulted in its commercial failure [3, 13].

For hybrid kernel designs, the most similar work to our proposal is FusedOS [22]. FusedOS runs HPC applications on top of a lightweight kernel (ensuring very low OS noise), which in turn offloads system calls to a general purpose Linux kernel. Although the offloading mechanism is similar to the technique how the proposed segmentation kernel obtains system services from McKernel, FusedOS doesn't address the TLB issue. Another interesting OS design which provides specialized kernel for certain applications was proposed in Libra [5]. Libra provides an execution environment specialized for IBM's J9 JVM using a hypervisor partition, which transparently relies on an instance of Linux in another hypervisor partition to provide a networking stack, a filesystem, and other OS services.

6. CONCLUSION AND FUTURE WORK

In this paper, we have designed and implemented a hybrid OS kernel, where some CPU cores run on paging and rest utilize segmentation over Intel's Xeon Phi coprocessor. Running applications over segmentation can completely eliminate the cost of paging, such as overhead of TLB misses and page walks as well as the energy consumption associated with TLB support. In our design, applications executed on the segmentation kernel can transparently obtain OS services from dedicated CPU cores that execute in paging setup, and thus, can take advantage of the rich features of paging during system call execution.

Although our current implementation is restricted to 32-bit mode, we have successfully demonstrated that segmentation can provide superior performance compared to both regular and large page based virtual memory. For instance, running Graph500 on top of our segmentation design can yield up to 81% and 9% improvement compared to utilizing 4kB and 2MB pages in Intel's MPSS Linux, respectively. Furthermore, our experiments also showed that rich features of paging are unnecessary for HPC applications. Most importantly, we believe that hardware vendors of the x86 architecture should indeed consider full support for segmentation in 64-bit execution mode due to its potential benefits to the HPC community.

In the future, we intend to provide support for multithreading in the segmentation kernel and further evaluate applications not only from a performance perspective but also in terms of OS noise.

Acknowledgment

This work has been partially supported by the CREST project of the Japan Science and Technology Agency (JST) and by the National Project of Ministry of Education, Culture, Sports, Science and Technology (MEXT) called Feasibility Study on Advanced and Efficient Latency Core Architecture.

7. REFERENCES

[1] Graph 500 Benchmark. http://www.graph500.org/specifications.

[2] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[3] Intel iapx 432. http://en.wikipedia.org/wiki/Intel_iAPX_432.

[4] libhugetlbfs. http://libhugetlbfs.sourceforge.net/.

[5] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., and Wisniewski, R. W. Libra: A Library Operating System for a Jvm in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 44–54.

[6] Arcangeli, A. Transparent hugepage support. In *KVM Forum* (2010).

[7] Barr, T. W., Cox, A. L., and Rixner, S. Spectlb: a mechanism for speculative address translation. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on* (2011), IEEE, pp. 307–317.

[8] Basu, A., Gandhi, J., Chang, J., and Swift, M. D. H. M. M. Efficient virtual memory for big memory servers. *ISCA* (2013).

[9] Bensoussan, A., Clingen, C. T., and Daley, R. C. The multics virtual memory: concepts and design. *Communications of the ACM 15*, 5 (1972), 308–318.

[10] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGARCH Computer Architecture News 36*, 1 (2008), 26–35.

[11] Bhattacharjee, A. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 383–394.

[12] Borkar, S. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference* (2007), ACM, pp. 746–749.

[13] Colwell, R. P., Gehringer, E. F., and Jensen, E. D. Performance effects of architectural complexity in the intel 432. *ACM Transactions on Computer Systems (TOCS) 6*, 3 (1988), 296–339.

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

[15] Intel Corporation. *INTRODUCTION TO THE iAPX 432 ARCHITECTURE*, 1981.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software DeveloperAfs Manual*, Sep 2013.

[17] Kahn, K. C., Corwin, W. M., Dennis, T. D., D'Hooge, H., Hubka, D. E., Hutchins, L. A., Montague, J. T., and Pollack, F. J. imax: A multiprocessor operating system for an object-based computer. *ACM SIGOPS Operating Systems Review 15*, 5 (1981), 127–136.

[18] McCurdy, C., Cox, A. L., and Vetter, J. Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on* (2008), IEEE, pp. 95–104.

[19] Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 89–104.

[20] Nayfeh, B., and Olukotun, K. A single-chip multiprocessor. *Computer 30*, 9 (1997), 79–85.

[21] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

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

[23] Pham, B., Vaidyanathan, V., Jaleel, A., and Bhattacharjee, A. Colt: coalesced large-reach tlbs. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on* (2012), IEEE, pp. 258–269.

[24] Ranganathan, P. From microprocessors to nanostores: Rethinking data-centric systems (vol 44, pg 39, 2010). *COMPUTER 44*, 3 (2011), 6–6.

[25] Romanescu, B. F., Lebeck, A. R., Sorin, D. J., and Bracy, A. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010), IEEE, pp. 1–12.

[26] Si, M., and Ishikawa, Y. Design of Direct Communication Facility for Many-Core based Accelerators. In *CASS'12: The 2nd Workshop on Communication Architecture for Scalable Systems* (2012).

[27] Sodani, A., and Processor, C. A. M. Race to exascale: Opportunities and challenges. In *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture* (2011).

[28] Talluri, M., and Hill, M. D. *Surpassing the TLB performance of superpages with less operating system support*, vol. 29. ACM, 1994.

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