Extending and Benchmarking the “Big Memory” Implementation on Blue Gene/P Linux

Kazutomo Yoshii, Harish Naik, Chenjie Yu, Pete Beckman
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA
{kazutomo,hnaik,cjyu,beckman}@mcs.anl.gov

ABSTRACT

Despite the fact that Linux is a popular operating system for high-performance computing, it does not ensure maximum performance for compute-intensive workloads. In our previous work we presented “Big Memory”—an alternative, transparent memory space that successfully removes the memory performance bottleneck on Blue Gene/P Linux. The initial Big Memory worked only as a per node resource. In this work we extend it to a per core resource and describe the details of the implementation. We evaluate our new implementation by running various benchmarks and the Nek5000 application. Compared with IBM’s Compute Node Kernel that is noise-free and lightweight, the Nek5000 application runs with only 1.2% performance loss on Linux with Big Memory at 32 K cores. Our benchmark results show no significant performance degradation from OS noise caused by Linux at a scale of up to 32 K cores, although irregular OS events are still present.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements*; D.4.2 [**Operating Systems**]: Storage Management—*Main memory*; C.5.1 [**Computer System Implementation**]: Large and Medium (“Mainframe”) Computers—*Super (very large) computers*

Keywords

Blue Gene, Linux, compute node, OS noise, TLB, memory management, CPU affinity, NPB, multi-core

1. INTRODUCTION

Linux is possibly the most popular operating system (OS) today, used in embedded devices, smart phones, personal computers, servers, mainframes, and high-performance computing (HPC). Linux was originally designed to handle multiuser, multitasking workloads in PCs and servers. Different platform have unique characteristics and therefore require tuning. On smart phones or PDAs, for example, power management and timer management are important. Linux as a server OS has to handle a large number of user processes and threads efficiently.

HPC is no exception and also has its own unique characteristics. HPC applications are very CPU or memory intensive and latency sensitive, so oversubscription significantly affects the performance [14]. Unlike in general purpose environments, only HPC applications and supporting processes such as job management or I/O-related daemons run in a HPC environment. Responsiveness of supporting processes is also important because of the latency sensitivity. A well-behaving scheduler needs to balance these two requirements; the default scheduler in the Linux kernel does not always do that, hampering the performance of HPC applications. The Linux scheduler also migrates processes for load balancing, which heavily impacts HPC application performance [12]; However we can mitigate the performance impact by using existing OS features. For example, UNIX traditionally allows users to change the scheduling priority, so that we can set HPC applications to the highest priority. Linux has system calls that enable users to change the scheduling policy and set the CPU affinity.

Like other general-purpose operating systems, the Linux kernel employs paged virtual memory, which provides various benefits, including process isolation, copy-on-write optimization, and simplicity of the physical memory allocation. However, the overhead from the memory management is considerable for HPC [18]. Paged virtual memory drastically degrades memory access performance as a result of page faults and translation lookaside buffer (TLB) misses, and it uses additional memory for page table maintenance. Contiguity of physical addresses is not guaranteed with virtual addresses crossing page boundaries. Furthermore, pages can be remapped, so a physical address associated with a virtual address is not guaranteed to be constant. This feature is particularly problematic for direct memory access(DMA) units that can deal only with physical addresses, so a software layer must keep track of page mappings and convert virtual addresses to physical ones. The overhead from this activity can be enormous.

Another issue of Linux is OS noise. Linux is a tick-based kernel, which periodically interrupts running tasks for a very short time. Studying the relationship between OS noise [1,3,8,9,13,19,20,22] and the scalability of large-scale applications is a major research topic. In our later work [23] we showed that for a certain class of applications like the Parallel Oceans Program, a tick based kernel like Linux provides a better performance.

In order to avoid the performance problems, some high-end machines such as IBM Blue Gene use dedicated light-weight kernels. IBM's Compute Node Kernel (CNK) [10], which is the default OS for Blue Gene/P (BGP), is not a tick-based kernel and lacks multiuser support, time-sharing, and multitasking. Each CNK process is pinned down to a CPU core and never migrates. CNK allows the spawning of additional user threads other than the main thread, but context switch happens only at system calls that internally yield the CPU. CNK does not employ paged virtual memory but provides a simplified, offset-based mapping from physical memory to a process's virtual address space. TLB entries are statically installed to map kernel space, user space, and some memory-mapped I/O (MMIO) at boot time. Thus, there is no memory overhead. Basically, HPC applications enjoy the native performance of the hardware. Unfortunately, the simplicity of the design is also an obstacle; it brings an inflexibility and a lack of features and capabilities. This situation prompted us to replace CNK with a Linux kernel as a part of the ZeptoOS project [25]. Our aim is to create a fully open software stack that enables independent computer science research on massively parallel architectures, enhances community collaboration, and fosters innovation.

In our previous work we presented "Big Memory" [24]—an alternative, transparent memory space that successfully eliminates the memory performance bottleneck on Blue Gene Linux. Our initial Big Memory implementation allowed users to run only a single Big Memory process per node; users had to create additional threads if they wanted to utilize all the CPU resources. Here we discuss the extending of Big Memory to a per core resource and the adding of strict CPU affinity.

2. IBM BLUE GENE/P

The IBM Blue Gene/P (BGP) architecture [15] was introduced in 2007 to replace the original Blue Gene/L design [17]. Blue Gene racks consist of two kinds of nodes: compute nodes, running the application code, and I/O nodes, responsible for system services such as file I/O. Compute nodes and I/O nodes use the same system-on-a-chip (SoC) design that consists of four PowerPC 450 cores, network devices, etc.

The PowerPC 450 core is a 32-bit design with SMP support, running at 850 MHz. Each processor core has a dual-pipeline floating-point unit with *fused multiply-add* (FMA) instructions. The peak floating-point performance of the whole CPU is 13.6 Gflops. Each core has a 32 kB[1] L1 instruction cache and a 32 kB L1 data cache (cache coherence is maintained with a write-invalidate protocol). The peak fill rate is 6.8 GB/s, with a latency of 4 CPU cycles. The L2 cache is smaller than the L1 and serves as a stream prefetching buffer. The CPU has a common 8 MB L3 cache with a latency of approximately 50 CPU cycles. Nodes have either 2 GB or 4 GB of main memory. The main store bandwidth is 13.6 GB/s, with a latency of approximately 100 CPU cycles.

Linux runs on I/O nodes to handle I/O requests from associated compute nodes. IBM provides a patch that enables PowerPC 44x Linux kernel to run on the BGP nodes: since the base kernel lacks SMP support, interprocess-interrupt (IPI) is added; FPU context is extended to store BGP FPU registers; a special console driver is implemented to forward kernel message via mailbox; and an ethernet driver is added to enable the BGP 10 GbE hardware.

Regular 32-bit PowerPC executables work on the BGP Linux kernel; however, code compiled specifically for BGP using a patched GNU C compiler might not work on other 32-bit PowerPC processors because of the custom BGP FMA instruction set.

The I/O node Linux kernel can be booted on compute nodes, since the main processor core is the same. However, modifications are required to make it useful; Root filesystem is required to perform boot-time initialization. We created a dedicated small ramdisk that contains the root filesystem for compute nodes. After a basic node initialization, it invokes the ZOID [16] daemon, which is responsible for job control (spawning and termination of application processes), forwarding of standard input and output streams to the control system, IP forwarding between compute nodes and I/O nodes, and file I/O forwarding to the I/O nodes (directly or via FUSE), since compute nodes cannot access storage resources directly.

3. HIGH PERFORMANCE MODE

Initially, there existed a large gap between IBM CNK and Linux in the memory access performance, even after we increased the system page size under Linux to 64 kB from the default size of 4 kB. Linux suffers from TLB misses, where none occur under CNK since it statically installs large TLB entries to cover the whole address space of a user program. In our previous study [23, 24] we observed that, in the worst case, Linux obtained approximately one-third of the memory bandwidth of CNK (and only one-twelfth with 4 kB pages).

Hugetlb can mitigate the memory performance issue. However, *hugetlbfs* does not eliminate TLB misses completely, so for highly irregular memory access patterns significant performance losses can still occur. Another problem of hugetlbfs is that it is not transparent, requiring additional programming. Transparent support for huge pages has been introduced in kernel 2.6.38. However, it is currently limited; for example, it is not supported for file-backed regions such as application text and data sections.

In our earlier work, we presented the implementation of "Big Memory"—an alternative, transparent memory space. The idea was to reserve a dedicated physical memory region at boot time and statically install TLB entries at page fault time to cover the Big Memory region. The TLB entries remain in place until the process is scheduled out again. Also, the Linux *exec* handler transparently prepares Big Memory when a special binary is about to be loaded. We verified this result on 1,024 nodes of Blue Gene/P using the NAS Parallel Benchmarks and found the performance on Linux with Big Memory to fluctuate within 0.7% of CNK.

In the initial implementation of Big Memory, only one Big Memory process per node, which is known as the SMP mode, was supported. To take advantage of all CPU cores on the node, the user had to create multiple threads within the Big Memory process. This was inconvenient for common MPI-only jobs, so we decided to extend Big Memory to support four Big Memory processes per node, which is known as the

[1]Throughout this paper, we use kB, MB, or GB in the context of memory size; 1 kB equals 1,024 bytes.

virtual node mode.

The basic idea of our extension of Big Memory is to strictly bind a Big Memory process to particular core and never allows the process to migrate when it runs in the virtual node mode. Processor binding or processor affinity [2] is a well-known technique to improve the performance of HPC applications; a userspace job launcher spawns a job, setting processor affinity to avoid over-subscription. We applied this idea into the kernel implementation so that the in-kernel Big Memory management codes can identify each Big Memory process by just reading the CPU core identification, which takes essentially one CPU cycle. This idea also keep the implementation simple.

To support the virtual node mode, we extended the Big Memory management code to be able to handle four instances of the memory management data structure. Also physical resource constraint is challenging. For example, organizing TLB entries efficiently is a challenge because PowerPC 450 has only 64 TLB entries per core. Paged memory, which under the ZeptoOS kernel can coexist with Big Memory, needs as many free TLB slots as possible; otherwise the performance will degrade. The Linux kernel itself reserves one to three TLB entries to cover the area where Linux kernel text and data sections are loaded. CNS requires another TLB entry to cover its text and data. BGP MMIO devices such as lockbox, UPC, tree, and DMA require several more TLB entries.

3.1 PowerPC 450 TLB

Unlike other processors, the PowerPC 450 TLB is fully managed by software. It is flexible: the structure of the page table entry (PTE) is left completely up to the software designer. It also allows one to statically install a TLB. On the other hand, the lack of hardware acceleration such as page-table walking is a disadvantage that results in large slowdowns with small page sizes.

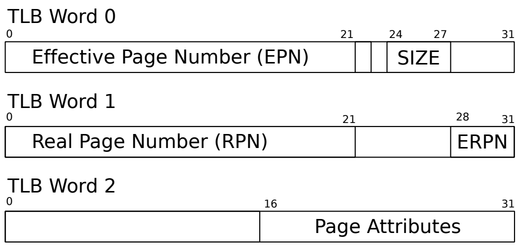

Figure 1: TLB Words

A TLB entry can be updated by using the *tlbwe* instruction. An entry is divided into three TLB registers: TLB Word0, Word1 and Word2 (see Figure 1). TLB Word0 is stores the virtual page address and the page size. Page sizes of 1 kB, 4 kB, 16 kB, 64 kB, 256 kB, 1 MB, 16 MB, 256 MB, and 1 GB are supported, and different page sizes can be used simultaneously. TLB Word1 stores the physical page address. TLB Word2 contains read, write, and executable permission for both supervisor and user mode, cache invalidation control, write-through flag and so forth. The *tlbre* instruction is used to read a TLB entry.

Figure 2 illustrates how a virtual address is converted into a physical one. On a PowerPC 450, technically, the virtual address is 41 bits long, and the physical address is 36 bits long. TID is not currently used, so the effective address, which is calculated from storage instructions, is essentially the virtual address. The bit size of the offset is $\log_2(pagesize)$. Note that the start address of both the virtual and physical page must be aligned to the page size. Mapping the virtual address 0x10000000 into the physical address 0x20000000 with a 256 MB page, for example, is valid; but mapping the same addresses with a 1 GB page, for example, is invalid. There is little chance of using 1 GB pages on a 32-bit architecture.

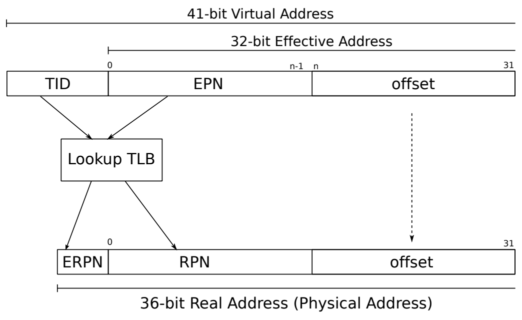

Figure 2: TLB Lookup

3.2 Physical Memory Allocation

The physical memory area used in Big Memory is reserved at boot time so that we can avoid complicated physical memory management. The size of Big Memory can be specified with Linux kernel parameters. The Linux kernel cannot use the reserved memory for regular paging; our memory system is targeted at HPC applications. BGP compute nodes are normally rebooted between jobs, so the reservation will be reset for the next job.

The physical memory area used by Big Memory is adjacent to the *lowmem* region where Linux kernel text, data, and runtime kernel data structures reside. To reserve the physical memory region, we modified the *MMU_init()* function, an early-stage physical memory setup code, to reduce the detected DRAM size by the specified size of Big Memory. We also had to write a code that relocates the CNS into a preallocated section within the kernel lowmem region because the CNS was originally loaded at the end of DRAM, preventing us from using a large TLB near the end of the memory space.

To maximize the performance of the DMA unit and simplify the software layer, we ensure that contiguous virtual addresses are also contiguous in the physical memory. Moreover, TLB page addresses must be aligned to the TLB page size; for example, 1 GB TLB must be aligned to the 1 GB boundary.

The Big Memory area is divided by the number of compute node processes, which is determined by the running mode. In our previous implementation, only the SMP mode (one task per node) was supported; no physical memory partitioning was required. In the virtual node mode, for example, the Big Memory area is partitioned into four areas, each assigned to one core.

3.3 ELF Flag and Kernel ELF Loader

Having an explicit, custom API to use a resource is frequently a hassle. Transparency is important, so we decided to alter the ELF header of application binaries and modify the Linux kernel ELF binary interpreter, specifically, the `load_elf_binary` function, which is invoked from the `execve` system call.

We use the `e_flags` field in the ELF header, which is reserved for processor-specific data. More specifically, we use the 24th bit[2] in the `e_flags` field. We refer to the executables with the flag set as ZeptoOS Compute Binaries, or ZCBs.

The `load_elf_binary` function examines the ELF header to see whether the binary being loaded is a ZCB. If it is, the function requests a new Big Memory process. If the request succeeds, the kernel sets a bit in the `personality` field in the task structure so that other kernel functions can easily determine, with minimum runtime overhead, that the process is a ZCB by accessing the `current` variable. Once the flag is set, the Big Memory region becomes available; however, the first access attempt will result in a page fault (see Section 3.4).

The Big Memory loader constructs the initial stack frame that contains the command line arguments, environment variables, and auxiliary vectors. Unlike a regular process, the contents of the initial stack is copied into the Big Memory region at `exec`. Similarly, the text and data sections of the program are also loaded into it. The next step is to install a virtual memory area (VMA) that covers all Big Memory address ranges to prevent the Linux kernel from invading the virtual address ranges that are assigned to Big Memory.

We added a memory manager to the Linux kernel to keep track of the memory chunks for `mmap` requests in Big Memory, which utilizes the kernel's red-black tree—a structure normally used for managing VMAs. The red-black tree is a self-balancing search tree, which can be searched in $O(\log n)$ time, where n is the total number of elements in the tree. In our previous implementation, Big Memory only supported one instance. We extended the memory manager to support up to four instances to support the virtual node mode.

While VMAs are installed to cover the Big Memory virtual address ranges, no PTE is installed for the ranges.

In the virtual node mode, the ZOID control process forks the application processes with the CPU affinity to match the BGP rank mapping. The affinity is preserved across the `execve` system call; once a ZCB binary is started, the `sched_setaffinity` system call is disabled for that process so that it cannot migrate away to another core. Setting CPU affinity is not needed in the SMP (one task per node) mode.

3.4 Page Fault

The rounded rectangle in Figure 3 denotes the extension that we have added for Big Memory handling. If a PTE is found for a faulting address, the handler simply fills in a TLB entry from the PTE. This situation never happens for the Big Memory region since Big Memory does not have an associated PTE. If no PTE is found, our extension handler is invoked. It checks whether the current task is a ZCB and whether the faulting address is within the Big Memory virtual address ranges. If so, it installs the TLB entries that cover the Big Memory region. Note that a set of TLB entries is chosen based on the faulting CPU core, since the Big memory process never migrates in the virtual node mode. The Big Memory TLB entries stay installed until the process is scheduled out; all the TLB entries except kernel lowmem and MMIO TLB entries are invalidated at the context switch. Within a process context, no TLB misses occur.

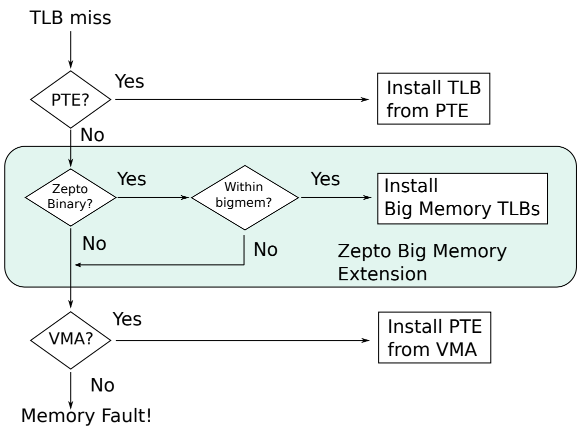

Figure 3: Big Memory Page Fault Handler

4. PERFORMANCE EVALUATION

4.1 Node Characteristics

To evaluate single node characteristics, we ran our OS noise measurement benchmark Selfish [3] and a random memory access benchmark.

The single-node Selfish benchmark reported a 0.05% OS noise ratio (see the upper graph in Figure 4) on a 2.6.29-based Linux kernel, which is the base kernel for our current work. Previously we obtained approx. 0.03% from the benchmark on a 2.6.19-based Linux kernel; the OS noise ratio has thus slightly increased.

We extended the benchmark to be able to capture the OS noise events in a parallel fashion. We ran it in the virtual node (four tasks per node) mode and found that the OS noise is unbalanced between cores (see the lower graph in Figure 4). As the figure shows, one of the cores has significantly different behavior. Core 1 experiences ten times higher OS noise than do the other cores: 0.53% noise ratio, with 0.05% on the remaining cores; the average OS noise ratio is 0.17%. We have identified this additional OS noise to be caused by the BGP kernel console driver.

Our random memory access benchmark first allocates an array and sets a random number in each array element. Then it reads an array element and use that value as an index for the next element.

We ran the benchmark in two modes: binding and non-binding. Binding means that the processor affinity mask is explicitly set for each MPI task so that the tasks run on separate cores and process migration never occurs. Non-binding means that the default process scheduling policy of the Linux kernel is in use, so migrations can occur. Note that Big Memory requires the use of binding.

Figure 5 shows the results obtained by running the benchmark on all four cores under CNK, Linux with 64 KB pages, and Linux with Big Memory. We confirmed that there is essentially no performance gap between Linux with Big Memory and CNK. The memory performance on Linux with 64 KB pages dramatically drops after 4 MB because of TLB

[2]PowerPC bit ordering

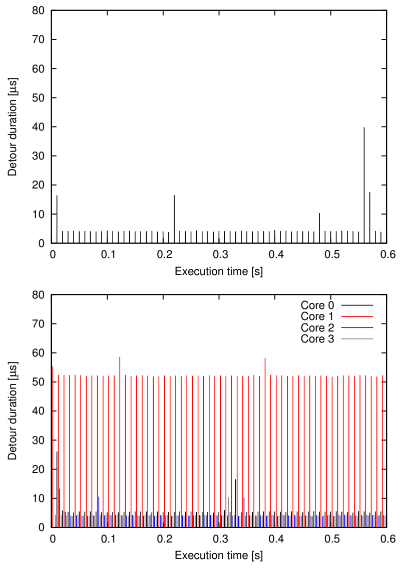

Figure 4: OS noise with the ZeptoOS compute node Linux kernel(upper), and OS noise per core(lower)

misses; the maximum memory area that can be covered by 64 KB TLB entries is slightly less than 4 MB because several TLBs are reserved for kernel lowmem mapping, MMIO, and other system purposes. Also, low performance in the nonbinding mode shows that the Linux process scheduler can be a source of significant performance problems if left unchecked.

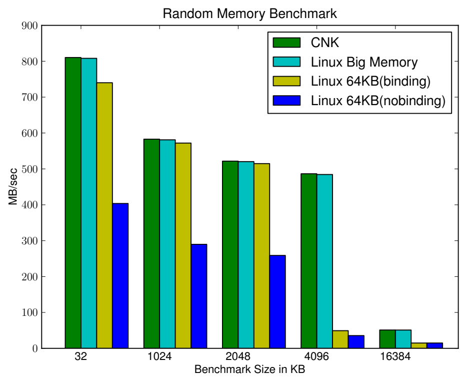

Figure 5: Memory Benchmarks: CNK vs Linux with Big Memory vs Linux with 64 KB page

4.2 MPI Microbenchmark

We ran a custom MPI microbenchmark to compare CNK and Linux with Big Memory in the virtual node mode.

Figure 6, top, shows the results of MPI latency measurements. With MPI Barrier, the latency gap between CNK and ZeptoOS slightly increases as the total core count increases. We believe the OS noise contributes to this, but the effect is minor and is unlikely to be observed in real applications (as they do not normally invoke MPI Barrier 400,000 times per second). In the MPI Send/Recv ping-pong latency benchmark, Linux with Big Memory performs slightly better than CNK. The message size is 4 bytes. The performance is due to the fact that the low-level userspace library, which converts the virtual to physical memory addresses for the DMA unit, has less overhead under ZeptoOS than under CNK. Again, this is unlikely to be a significant effect in real applications.

Figure 6, bottom, shows the bandwidth measurements. In both the Send/Recv and Broadcast benchmark, the performance gap between ZeptoOS and CNK is small (0.1–0.5%) and does not exhibit any scalability degradation. The individual message size is 16 MB. Both communication methods use the torus DMA unit, which offloads the CPU, making the communication less sensitive to noise. With Allreduce, the performance gap is slightly larger (0.9–1.7%), but again there are no signs of any scalability problems. AllReduce uses the collective network, which does not have the DMA feature, so the CPU has to take care of copying the message packets from main memory to the network device. Thus, AllReduce is more noise-sensitive than is Send/Recv or Bcast.

4.3 NAS Parallel Benchmarks

We ran NAS Parallel Benchmarks (NPB) version 3.3 to compare Linux with 64 kB pages, Linux with Big Memory, and CNK. We chose the EP and FT benchmarks. EP (Embarrassingly Parallel) generates independent Gaussian random variates using the Marsaglia polar method and does not invoke MPI primitives during computation. FT performs a fast fourier transform to solve a three-dimensional partial differential equation and heavily depends on MPI_Alltoall, which could cause scalability problems. We used the same IBM XL compiler to compile source code in all cases.

Figure 7 shows the results from experiments with class C problem size, comparing two Linux memory models. In a strict sense, this comparison is not accurate because not only the memory models but also the communication stacks used are different. With Big Memory, we can use highly tuned BGP communication libraries originally written for CNK. These do not work with paged memory, however, and fixing them would be highly nontrivial. Instead, we used a Linux Ethernet driver built on top of the torus network, developed by IBM, and ran a stock MPICH library configured for TCP/IP. We ran with MPICH in two modes: binding and nonbinding. Even with EP, which is not communication-intensive, Linux with Big Memory shows a significant advantage over paged memory. As we observed in the random access benchmark result, the nonbinding benchmark performs poorly.

Figure 8 shows the results from experiments with class E problem size, comparing CNK and Linux with Big Memory. With EP, Linux is slower than CNK by 1.4–1.9%. EP is a CPU-intensive program and stresses the memory system. We suspect that Linux timer interrupts and other kernel activities thrash the L1 cache, resulting in these differences. With FT, Linux is faster than CNK by 0.2–3.9%. We assume that, just like with earlier latency measurements, the efficiency of the userspace address translation layer implementation under Linux contributes to this gain. In either cases, there are no signs of any scalability problems.

4.4 Nek5000

Nek5000 is a mature DNS/LES computational fluid dynamics solver developed at the Mathematics and Computer

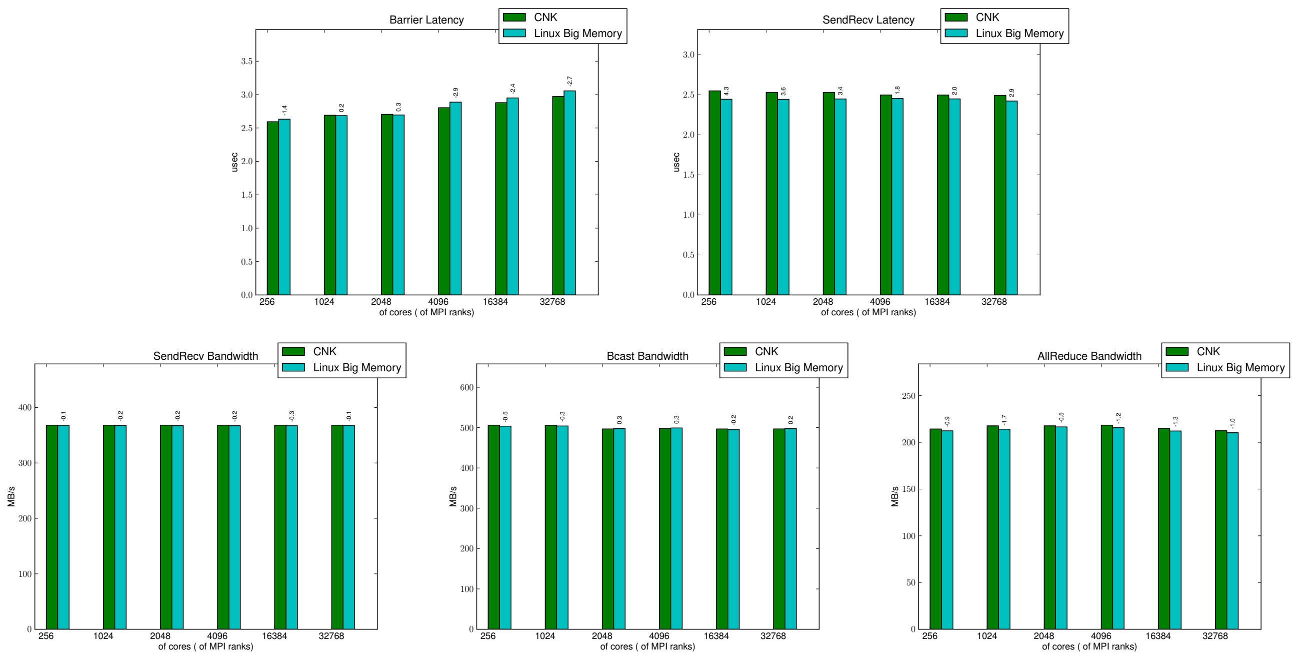

Figure 6: MPI Microbenchmarks: CNK vs Linux with Big Memory

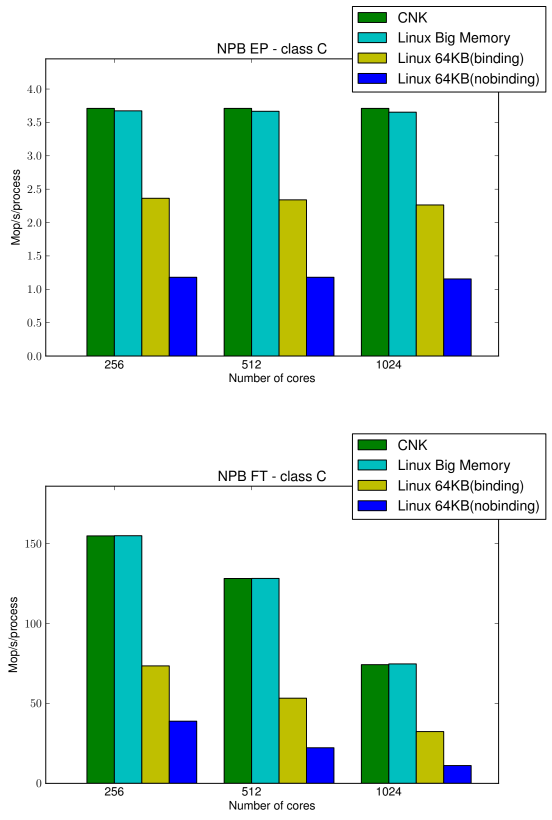

Figure 7: NPB: Linux with 64 KB pages vs Linux with Big Memory

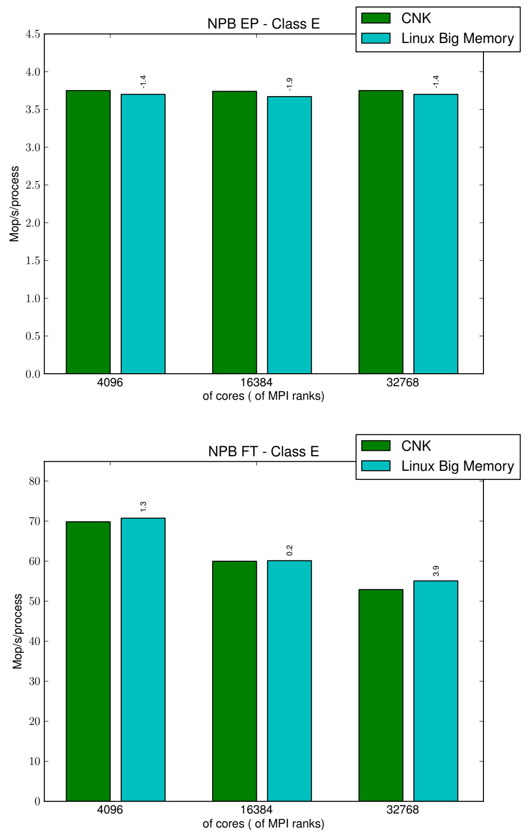

Figure 8: NPB: CNK vs Linux with Big Memory

Science Division of Argonne National Laboratory. Nek5000 simulates unsteady incompressible fluid flow with thermal and passive scalar transport. It can handle general two- and three-dimensional domains described by isoparametric quad or hex elements. In addition, it can be used to compute axisymmetric flows. It is a time-stepping-based code and does not currently support steady-state solvers, other than steady Stokes and steady heat conduction.

Figure 9 shows the results from Nek5000 of fluid flow simulation in a T-Junction, comparing CNK and Linux with Big Memory. The Nek5000 application runs with only 1.2% performance loss on Linux with Big Memory at 32 K cores.

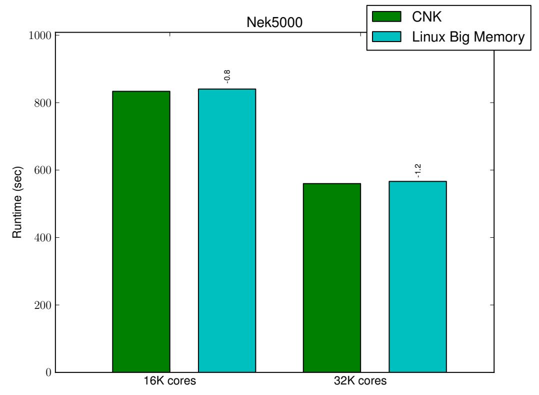

Figure 9: NEK: CNK vs Linux with Big Memory

5. RELATED WORK

Linux provides support for large memory pages through the *hugetlbfs* [4] mechanism. Using these pages dramatically reduces the number of TLB misses, improving performance. However, this feature is not transparent—applications need to invoke the `mmap` system call explicitly to make that memory available, and by then it is too late to use the memory for segments such as application text, heap, or stack.

Shmueli et al. [21] evaluated Linux on the compute nodes of Blue Gene/L and identified TLB misses as a major source of node-level performance degradation. To mitigate the problem, they used hugetlbfs. They employed libhugetlbfs [11], a wrapper library that semi-transparently maps an application's text, data, and heap to a memory area backed by hugetlbfs. Their approach allowed Linux to achieve a performance comparable to CNK, both at the node level and systemwide. However, hugetlbfs does not eliminate the TLB misses completely, so they can still be a performance problem for some applications. This approach also does not help with programming the DMA engine on BGP. Moreover, the approach requires dynamic linking, while on Blue Gene almost all executables are statically linked, since this is the compiler default on that platform. The authors also found that dynamic linking introduced an overhead when accessing floating-point constants.

Ferreira et al. [9] did an empirical study of HPC applications that are sensitive to OS noise and the impact system design parameters have on them. The work examined the effects of varying the ratio of peak network bandwidth to compute performance and showed how it influences OS noise.

De et al. [5] identified that about 63% of operating system jitter came from timer interrupts. The rest of the jitter was shown to originate from other, mostly nonessential operating system services. They introduced a tool to identify new sources of operating system jitter. In their later work [6] they show that by synchronizing jitter, one can significantly reduce performance degradation. They also introduce a system that can predict scalability based on jitter traces and other system latency measurements [7].

6. CONCLUSIONS

Memory management overhead and process scheduling policies remain an obstacle on the road toward successful adoption of Linux on large-scale massively parallel systems. This paper described how we incorporated high-performance mode into Linux kernel while keeping other Linux features intact. We presented the implementation details and latest performance data for Big Memory—an alternative, transparent, high-performance memory space that we recently reimplemented as a per core resource, enabling the execution of up to four MPI tasks per Blue Gene/P compute node. We compared Big Memory with regular Linux paged memory as well as with the memory architecture of IBM's lightweight kernel CNK. The results show that our ZeptoOS Linux kernel nearly matches the performance of the CNK. We did not observe any significant performance degradation from OS noise caused by Linux in communication and computation benchmarks at a scale of up to 32 K cores, although irregular OS events are still present, including order-of-magnitude differences in OS noise ratio between cores. We also found that the efficiency of the virtual-to-physical address translation for the DMA engine can impact the performance of some workloads, giving our implementation under Linux a slight advantage.

We are entering a massive multicore processor era. Irregular OS noise like what we have observed might impact the node scalability of future systems. We will continue to study the performance of Linux with larger and more complex HPC applications on existing systems and will work to identify scalability bottlenecks on emerging and future platforms. Re-designing the kernel mechanisms as we find problems will continue to be part of our future research.

Acknowledgments

We thank IBM's Todd Inglett, Thomas Musta, Thomas Gooding, George Almási, Sameer Kumar, Michael Blocksome, Blake Fitch, Chris Ward, and Robert Wisniewski for their advice on programming the Blue Gene hardware. We also thank our past summer interns Balazs Gerofi, Kazunori Yamamoto, Peter Boonstoppel, Hajime Fujita, Satya Popuri, and Taku Shimosawa, who contributed to the ZeptoOS kernel. Additionally, we thank Astron's John W. Romein and Chris Broekema who contributed to the Big Memory implementation. Finally, we thank other members of the ZeptoOS team, in particular Kamil Iskra, for their contributions to the project.

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory.

7. REFERENCES

[1] Agarwal, S., Garg, R., and Vishnoi, N. K. The impact of noise on the scaling of collectives: A theoretical approach. In *Proceedings of the 12th International Conference on High Performance Computing* (2005), vol. 3769 of *Springer Lecture Notes in Computer Science*, pp. 280–289.

[2] Alam, S. R., Barrett, R. F., Kuehn, J. A., Roth, P. C., and Vetter, J. S. Characterization of scientific workloads on systems with multi-core processors. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2006), pp. 225–236.

[3] Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. Operating system issues for petascale systems. *ACM SIGOPS Operating Systems Review 40*, 2 (Apr. 2006), 29–33.

[4] Chen, K., Seth, R., and Nueckel, H. Improving enterprise database performance on Intel Itanium architecture. In *Proceedings of the Linux Symposium* (Ottawa, ON, Canada, July 2003), pp. 98–108.

[5] De, P., Kothari, R., and Mann, V. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *Proceedings of the IEEE International Conference on Cluster Computing* (2007), pp. 331–340.

[6] De, P., Kothari, R., and Mann, V. A trace-driven emulation framework to predict scalability of large clusters in presence of OS jitter. In *Proceedings of the IEEE International Conference on Cluster Computing* (2008), pp. 232 –241.

[7] De, P., and Mann, V. JitSim: A simulator for predicting scalability of parallel applications in presence of OS jitter. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I* (2010), pp. 117–130.

[8] Ferreira, K. B., Bridges, P., and Brightwell, R. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008).

[9] Ferreira, K. B., Bridges, P. G., Brightwell, R., and Pedretti, K. T. The impact of system design parameters on application noise sensitivity. In *Proceedings of the IEEE International Conference on Cluster Computing* (2010), pp. 146–155.

[10] Giampapa, M., Gooding, T., Inglett, T., and Wisniewski, R. W. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010).

[11] Gibson, D., and Litke, A. libhugetlbfs. `http://sourceforge.net/projects/libhugetlbfs`.

[12] Gioiosa, R., McKee, S. A., and Valero, M. Designing OS for HPC applications: Scheduling. In *Proceedings of the IEEE International Conference on Cluster Computing* (2010), pp. 78–87.

[13] Hoefler, T., Schneider, T., and Lumsdaine, A. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010).

[14] Iancu, C., Hofmeyr, S., Blagojevic, F., and Zheng, Y. Oversubscription on multicore processors. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing* (2010).

[15] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development 52*, 1/2 (2008), 199–220.

[16] Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), pp. 153–162.

[17] Moreira, J. E., et al. Blue Gene/L programming and operating environment. *IBM Journal of Research and Development 49*, 2/3 (Mar. 2005), 367–376.

[18] Moreira, J. E., et al. Designing a highly-scalable operating system: The Blue Gene/L story. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Tampa, FL, Nov. 2006).

[19] Nataraj, A., Morris, A., Malony, A. D., Sottile, M., and Beckman, P. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of the ACM/IEEE conference on Supercomputing* (2007).

[20] Petrini, F., Kerbyson, D. J., and Pakin, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Phoenix, AZ, Nov. 2003).

[21] Shmueli, E., Almási, G., Brunheroto, J., Castaños, J., Dózsa, G., Kumar, S., and Lieber, D. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *Proceedings of the 22nd ACM International Conference on Supercomputing* (Kos, Greece, July 2008), pp. 165–174.

[22] Tsafrir, D., Etsion, Y., Feitelson, D. G., and Kirkpatrick, S. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th International Conference on Supercomputing* (Cambridge, MA, June 2005), pp. 303–312.

[23] Yoshii, K., Iskra, K., Naik, H., Beckman, P., and Broekema, P. C. Performance and scalibility evaluation of "Big Memory" on Blue Gene Linux. *International Journal of High Performance Computing Applications 25*, 2 (May 2011).

[24] 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 International Conference on Parallel Processing Workshops* (2009), pp. 65–72.

[25] ZeptoOS project. `http://www.zeptoos.org/`.