Characterizing the Performance of "Big Memory" on Blue Gene Linux

Kazutomo Yoshii[1], Kamil Iskra[1], Harish Naik[1], Pete Beckman[1,2]
[1]*Mathematics and Computer Science Division*
[2]*Leadership Computing Facility*
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA
Email: {kazutomo,iskra,hnaik,beckman}@mcs.anl.gov

P. Chris Broekema
ASTRON
Netherlands Institute for Radio Astronomy
Oude Hoogeveensedijk 4, Dwingeloo, The Netherlands
Email: broekema@astron.nl

***Abstract*—Efficient use of Linux for high-performance applications on Blue Gene/P (BG/P) compute nodes is challenging because of severe performance hits resulting from translation lookaside buffer (TLB) misses and a hard-to-program torus network DMA controller. To address these difficulties, we present the design and implementation of "Big Memory"—an alternative, transparent memory space for computational processes. Big Memory uses extremely large memory pages available on PowerPC CPUs to create a TLB-miss-free, flat-memory area that can be used for application code and data and is easier to use for DMA operations. One of our single-node memory benchmarks shows that the performance gap between regular PowerPC Linux with 4 KB pages and IBM BG/P compute node kernel (CNK) is about 68% in the worst case. Big Memory narrows the worst case performance gap to just 0.04%. We verify this result on 1024 nodes of Blue Gene/P using the NAS Parallel Benchmarks and find the performance under Linux with Big Memory to fluctuate within 0.7% of CNK.**

Originally intended exclusively for compute node tasks, our new memory subsystem turns out to dramatically improve the performance of certain I/O node applications as well. We demonstrate this performance using the central processor of the LOw Frequency ARray (LOFAR) radio telescope as an example.

Keywords: petascale, Blue Gene, OS kernel, Linux, compute node, I/O node, memory performance, TLB, NAS Parallel Benchmarks, LOFAR

I. Introduction

The Blue Gene architecture [1], [2], developed by IBM, is one of the most successful contemporary massively parallel computer architectures, thanks to a high-speed interconnect, a highly scalable design, and a very low power consumption compared to other supercomputers.

Blue Gene machines normally run a dedicated compute node kernel (CNK) [3] on the compute nodes. CNK is essentially a microkernel that supports only one user thread per CPU core and provides a simplified, offset-based mapping from physical memory to the virtual address space. This design keeps the kernel small and simple; more important, since Blue Gene lacks hardware to handle translation lookaside buffer (TLB) misses efficiently, it maximizes the memory access performance, as well as the floating-point performance. It also simplifies the programming of hardware devices, in particular the DMA engine discussed later.

Unfortunately, the simplicity of the design is also an obstacle; it brings an inflexibility and a lack of features that are generally taken for granted in more general-purpose operating system kernels, such as multitasking and time sharing. This situation prompted us to replace CNK with a Linux kernel as a part of the ZeptoOS project [4], in an effort to create a fully open software stack to enable independent computer science research on massively parallel architectures, enhance community collaboration, and foster innovation.

In previous publications related to ZeptoOS, we focused on operating system jitter [5], [6] and on I/O forwarding [7]. This paper focuses on the performance of our Linux-based ZeptoOS compute node kernel, with an emphasis on memory management.

In the remainder of this section, we outline the key characteristics of the Blue Gene hardware design and their consequences for the software stack (both CNK and the ZeptoOS one). We focus on the Blue Gene/P (BG/P) architecture [8] introduced in 2007 to replace the original Blue Gene/L design [2].

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. These two types of nodes feature the same hardware; however, the available interconnects are different. The primary interconnect available on the compute nodes is a 3D torus network, used for high-performance point-to-point communication. A collective network is also available for more sophisticated operations; it doubles as a file I/O backbone connecting compute nodes with I/O nodes. Unlike the compute nodes, which normally run CNK, I/O nodes run a Linux kernel. I/O nodes also have a 10 GbE link to connect them to file servers,

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.
LOFAR is funded by the Dutch government in the BSIK programme for interdisciplinary research for improvements of the knowledge infrastructure. Additional funding is provided by the European Union, European Regional Development Fund (EFRO) and by the "Samenwerkingsverband Noord-Nederland," EZ/KOMPAS.

1530-2016/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPPW.2009.35

login nodes, and the service node. The only practical means of communication between compute nodes and the outside world is through the I/O nodes, using I/O forwarding.

We chose Linux as the foundation of our ZeptoOS kernel for the compute nodes because of its openness and popularity, but our decision was made easier by the fact that Linux already ran on the nearly identical I/O nodes. Only minor changes to the I/O node kernel itself were needed to boot it on the compute nodes. However, a number of far-reaching changes were required to make it *useful*. For example, we had to replace the CNK-specific IBM I/O forwarding and job control software with ZOID [7]. ZOID enables access to remote file systems from the compute nodes running Linux, but it also provides an interactive login capability to the compute nodes from the outside, which proved invaluable in further software development.

Internally, BG/P nodes use PowerPC 450 CPUs—a quad-core, 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 (the latter featuring a snoop filter to provide cache coherency between cores). The peak fill rate is 6.8 GB/s. The L2 cache is smaller than the L1 and serves as a stream prefetching buffer. The CPU has a common 8 MB L3 cache. Nodes have either 2 GB or 4 GB of main memory. The main store bandwidth is 13.6 GB/s.

At the time of writing, PowerPC 450 is not a component available separately on the market; it can be purchased only as part of a Blue Gene/P system. Consequently, a stock Linux kernel does not have the support needed to make Linux boot on that CPU. However, starting with Blue Gene/P, IBM provides the necessary patches to enable Linux to work on this processor. Regular 32-bit PowerPC executables run well on a BG/P Linux kernel; however, executables compiled specifically for BG/P using a patched GNU C compiler might not work on other 32-bit PowerPC processors because of the custom BG/P FMA instruction set.

Because of its embedded systems origins, the processor has only 64 TLB entries per core. Even worse, TLB misses must be handled in software, inside the Linux kernel, with an average cost of approximately 0.3 μs. Hence, with the default PowerPC Linux page size of 4 KB, if the memory is accessed randomly, the working set of the program must not be larger than 256 KB before the performance significantly degrades. The processor itself supports various page sizes ranging from 1 KB to 1 GB. Pages of different sizes can be used simultaneously; unfortunately, the Linux kernel lacks the flexibility needed to take advantage of this feature. We have experimented with increasing the system page size to 64 KB; we show the results later in the paper. Unfortunately, this approach is not really an option on the I/O nodes because the legacy software running there (in particular, the GPFS file system client code) works only with 4 KB pages. CNK takes full advantage of the hardware and statically maps all the system memory using large TLB entries, thereby eliminating TLB misses.

Unlike its predecessor used in Blue Gene/L, the PowerPC 450 has a coherent L1 data cache; the PowerPC *load and reserve* instruction works between cores, and coherent pages are available. However, the `tlbsync` instruction is not supported, so the Linux kernel has to use an interprocess-interrupt (IPI) to synchronize software TLB management. The BG/P Linux kernel is configured for a virtual memory split of 3 GB user space and 1 GB kernel space, with HIGHMEM enabled by default. BG/P-specific additions primarily include various network drivers.

In Blue Gene/P, the torus network between the compute nodes has been enhanced with a DMA engine. The engine can deal only with physical addresses; a software layer has to translate virtual addresses to physical ones. This process is simple with IBM's CNK, where contiguous virtual addresses are also contiguous in physical space. Unfortunately, with paged memory used in Linux, this is not the case—the translation is a lot more complex, and fragmented physical address space seriously limits the size of DMA operations, hurting performance.

Given the problems caused by paged memory on BG/P, we decided to investigate an alternative memory management scheme.

II. Related Work

Linux does provide support for large memory pages, through the *hugetlbfs* [9] 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. [10] 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 also employed libhugetlbfs [11], a wrapper library that semi-transparently maps 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 BG/P. Moreover, the approach requires dynamic linking, while on BG/P almost all executables are statically linked,

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

since this is the compiler default on Blue Gene. The authors also found that dynamic linking introduced an overhead on accessing floating-point constants.

Navarro et al. [12] designed an effective transparent *superpage* management system that utilizes larger physical pages to reduce TLB misses and implemented it in FreeBSD on the Alpha processor. When a page fault occurs, the size of the superpage is chosen, and a set of contiguous page frames that covers the superpage is allocated from the buddy allocator. Fragmentation control and superpage promotion are part of their design. They evaluated their implementation with both benchmarks and realistic workloads and observed a 30% to 60% performance improvement. They targeted a relatively general-purpose usage, however, whereas we are focusing on high-performance applications on massive parallel machines.

III. Early Performance Evaluation

In this section, we present the results of a number of performance measurements we made on a mostly unmodified BG/P Linux kernel.

A popular misconception is that Linux would not be suitable as a compute node kernel because of a high level of operating system noise. We have disproved this notion on several older platforms [5], [6].

Figure 1 shows the results obtained by using our OS noise measurement benchmark Selfish [13] on BG/P. Essentially, the benchmark is a tight, busy loop that records anomalies in its execution time caused by OS interrupts (i.e., system noise). In this case, the Linux kernel used only 0.027% of the CPU time, leaving 99.963% of the cycles to the benchmark. The interruptions are small and predictable, making them fairly easy to control.

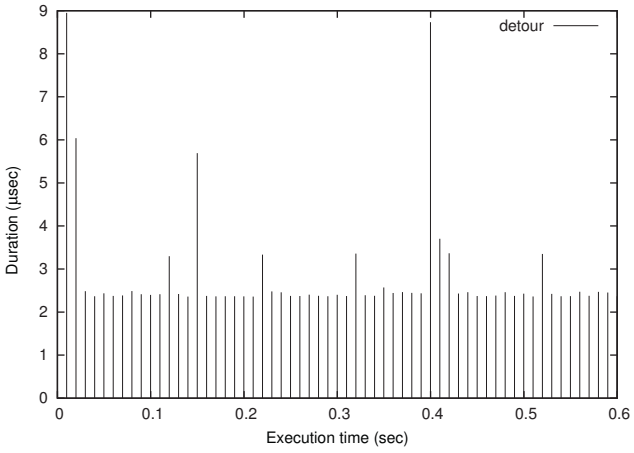

Figure 1. Noise on BG/P Linux

Table I also shows that Linux OS noise is low. The FPU benchmark is basically a tight loop of FMA instructions. The benchmark kernel easily fits in the L1 instruction cache. On IBM's CNK, the result matches the theoretical peak performance of BG/P floating-point unit per core (4 flops × 850 MHz yields 3.4 Gflops). Linux is only 0.03% slower—this number matches what we saw in our noise experiment.

Table I
Linux vs CNK: FPU benchmark

	Gflops	% of peak
CNK	3.400	100.00
Linux	3.397	99.97

Table II presents the results of one of our memory benchmarks. This benchmark reads data from a 128 MB memory region in a random fashion. Because of the problems with the TLB misses discussed earlier, Linux performs almost three times slower than the TLB-miss-free CNK. Increasing the system page size from 4 KB to 64 KB does improve performance, but it is still far worse than the CNK result. Clearly, a more radical solution is needed.

Table II
Random memory access benchmark

	MB/s
CNK	44.70
Linux 4 KB	14.39
Linux 64 KB	16.40

IV. Big Memory

We first briefly discuss standard paged memory management. We then explain our approach using "Big Memory."

A. Standard Linux Memory Management

The Linux kernel is a virtual memory operating system. The main purpose of virtual memory is process isolation, but virtual memory also provides other optimizations or functionality such as copy-on-write, file caching, and memory swapping. Nowadays, virtual memory is considered mandatory in general-purpose operating systems.

Figure 2 provides an overview of memory management in Linux. The address space of each process consists of a set of virtual address ranges called virtual memory areas (VMAs). VMAs are created when a new process is started; the `mmap` system call can also create a new VMA. Creating a VMA is not equivalent to physical memory allocation; this takes place only on the first memory access within the VMA. Memory access attempts outside the VMAs result in a memory fault. VMAs have access permissions associated with them; incorrect access attempts result in memory faults as well.

On most modern processors, user programs run inside virtual address spaces. In other words, user programs cannot address physical memory directly—the processor always has to convert a virtual address to the corresponding physical address. Page table entries (PTEs) are used for that purpose.

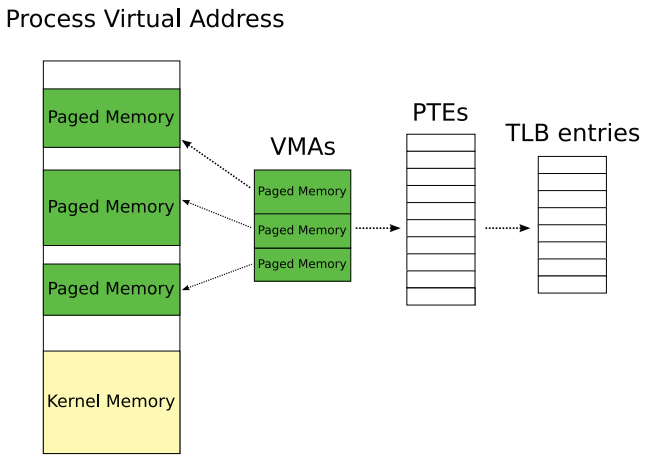

Figure 2. Memory Management in Linux

The operating system kernel is responsible for creating PTEs based on the VMAs and for storing them in memory in advance or upon request. PTEs are stored in the kernel space (in main memory), so accessing a PTE is simply an overhead. To alleviate this overhead, modern processors cache recent address conversions in the translation lookaside buffer. The TLB definitely improves memory access speed, but it is not a complete solution because TLB entries are scarce. When a TLB miss takes place, a TLB entry needs to be loaded from a PTE.

The TLB entries associated with paged memory are flushed by the Linux kernel when the kernel switches to another process or when an associated VMA is removed. PowerPC Linux statically pins down a few TLBs to cover kernel memory space, so TLBs associated with kernel memory are not flushed when switching context.

B. Our Approach

Our basic idea is to provide applications with special memory regions that are covered by larger pages; we refer to these regions as *Big Memory*. Unlike the hugetlbfs regions discussed in Section II, which use pages of intermediate size (2–4 MB) and only *reduce* the probability of TLB misses, Big Memory uses pages so large that the TLB misses are *eliminated* while the process is scheduled in.

We focus on the compute node environment here; we do not consider a general-purpose solution. The compute node environment has unique characteristics: a computational process tends to monopolize the CPU execution unit, floating-point unit, memory, network, and other resources. In order to achieve peak performance, a single computational thread per core is preferable, since the kernel has to save the current context (all CPU registers associated with the current process, including double-floating-point registers in the case of the PowerPC 450), load new context, and invalidate caches at context switching time, thereby incurring substantial overhead, especially for CPU intensive application. Our approach is to allow only a single computational process to run at a time, to let the Linux kernel pin down TLB entries to cover the Big Memory area on the first access after the computational process has been scheduled in, and to remove them when that process gets scheduled out (to keep the memory mapping private to the computational process). Thus, an application does not suffer TLB misses when it accesses its Big Memory area.

Physical memory needed for the Big Memory area is reserved at boot time and is thus not available for use by the kernel as a regular, paged memory. It can be used only by a special computational process. This dramatically reduces the complexity of the implementation.

Unlike the hugetlbfs-based solutions, our approach is fully transparent, requires no code changes to the application, works with static executables, and covers all the necessary application segments.

The Linux kernel automatically prepares the Big Memory area for a computational application. How does the kernel determine which process should use Big Memory? Our solution is to alter the application's executable file; we use the `e_flags` field in the ELF header, which is reserved for processor-specific data. We defined a custom flag and wrote a tool that toggles it. We refer to the executables with the flag set as Zepto compute binaries, or ZCBs.

C. ELF Binary Interpreter

To load a ZCB into Big Memory, we have modified the Linux kernel ELF binary interpreter, in particular, the `load_elf_binary` function, which is invoked from the `execve` system call.

First, the ELF header is examined to see whether the binary being loaded is a ZCB. If it is, the kernel sets a bit in the `personality` field in the task structure so that other kernel functions can easily determine that the process is a ZCB by accessing the `current` variable.

Then the kernel creates a virtual memory range for Big Memory, using a simple offset mapping. We currently use 256 MB pages to cover the application memory; in the future we will improve the granularity of the Big Memory area by using a combination of different page sizes.

Once the Big Memory mapping is initialized, the kernel temporarily installs the Big Memory TLB entries to copy the contents of both the command-line arguments and the environment variables to the application stack. The kernel also loads the application's text and data sections to Big Memory instead of using the usual file mapping; the Big Memory mapping cannot be used for file mappings because it bypasses the Linux page allocator (see below). In other words, the entire big memory area is populated when the application binary is loaded.

D. Memory Manager for Big Memory

Our kernel reserves one VMA to cover Big Memory, and our internal memory manager takes care of memory chunks

within the Big Memory area (heap, stack, etc.). To keep track of the memory chunks for `mmap` requests in Big Memory, the manager utilizes the kernel's red-black tree—a structure normally used for managing entire 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.

The ZCB process address space is actually hybrid; it contains both regular paged memory and Big Memory (see Figure 3). Note that the Big Memory VMA does not have any associated PTEs, since the physical addresses of memory in that region are fixed. The behavior of the `mmap` system call varies depending on the request type. Anonymous, private requests, as used for large C library `malloc` calls, go to Big Memory and are tracked by our internal memory manager. On the other hand, file-backed mapping requests—used, for example, to support shared libraries—simply go to the regular Linux paged memory manager because Big Memory cannot be used for file mappings.

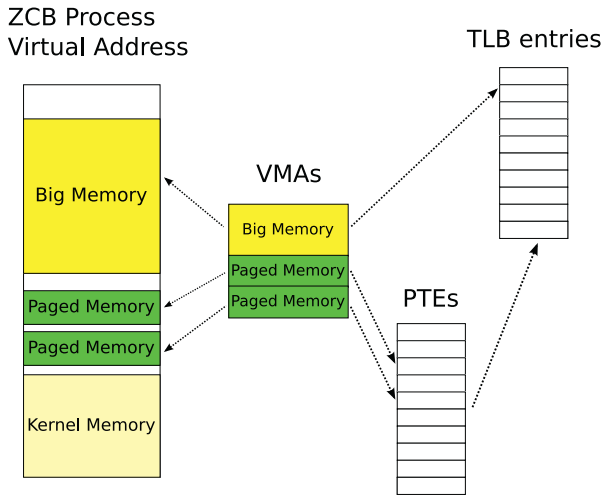

Figure 3. ZCB Process Address Space

E. Page Fault Handler

While we did not need to modify the TLB handler to implement Big Memory, we have added a hook to the Linux page fault handler (see Figure 4). The added code first checks whether the current task is a ZCB. If it is, the code checks whether the faulting address is within the Big Memory area; if so, it installs the Big Memory TLB entries. Essentially, we get a single TLB miss on the first access after the process has been scheduled in; the entries normally remain in place until the process is scheduled out again. With context switches being fairly rare on the compute nodes, the entries are semi-static.

As shown in Figure 5, our Linux kernel partitions TLB entries in four groups: kernel mapping, paged memory, Big Memory, and device mappings. The last two groups are

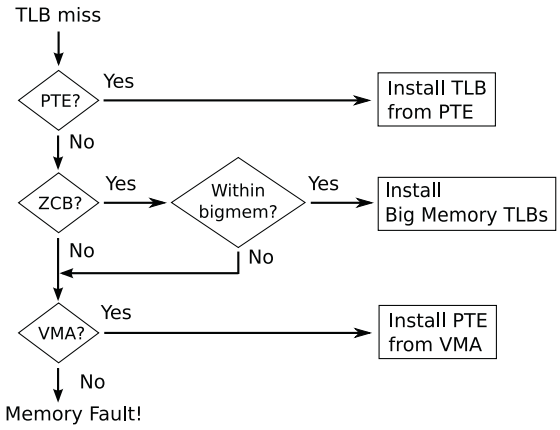

Figure 4. Big Memory Fault-Handling Flow

specific to our implementation. Currently, the number of TLB entries required by Big Memory is proportional to the area size; for example, seven entries are needed to cover 1792 MB. Three entries are used to cover the kernel memory (kernel low memory, to be precise). For efficiency, we also pin down some entries to cover BG/P-specific memory-mapped I/O devices: the collective network, the torus DMA, the lockbox, the Universal Performance Counter, and the Blue Gene Interrupt Controller. The Blue Gene Common Node Services code segment is also pinned.

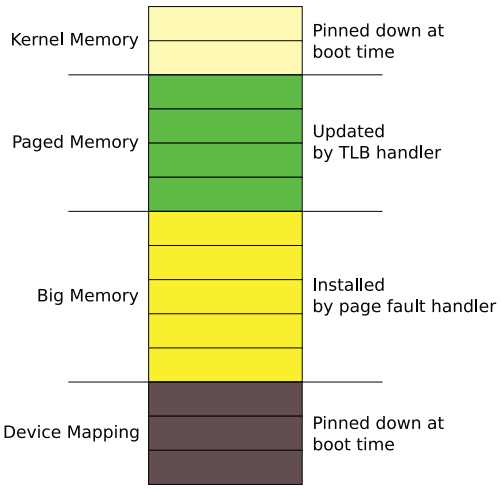

Figure 5. TLB Partitioning in the Zepto Kernel

F. Other Kernel Modifications

Introducing a new concept invariably causes issues; the Big Memory implementation is no exception. Big Memory breaks a number of assumptions in Linux memory management. A ZCB process is forked from regular Linux processes; however, it cannot call `fork` itself, since the fork code apparently depends on the copy-on-write technique. The lack of fork may be acceptable when we focus on a

high-performance compute node environment; besides, this is still better than the CNK, which does not support `fork` at all and must be rebooted to start another process.

Another issue is that the Big Memory mapping is strictly private and cannot be addressed from other processes' context. The Linux kernel function `access_process_vm` did not work for ZCB processes; this was an important issue because that function is used by the `ptrace` system call, on which debugging tools like *strace* and *gdb* depend. Luckily, the problem was relatively easy to fix; basically, we now temporarily install the Big Memory mapping while the function is running, to allow other processes to access the address space of the ZCB process.

V. Performance Evaluation

To evaluate single-node performance, we ran our memory microbenchmarks and an FFTW benchmark. For parallel performance evaluation, we used the NAS Parallel Benchmarks (NPB) [14]. The experiments were performed on BG/P compute nodes with four different kernels: IBM CNK, standard Linux with 4 KB pages, Linux with 64 KB pages, and our enhanced Linux with Big Memory support. We used the GNU compiler toolchain that supports PowerPC 450 instructions to compile the benchmarks. Program binaries are compatible between the kernels, with the exception of NPB. For NPB, we could run only on CNK and Linux with Big Memory because we do not have MPI support for paged memory. Slight differences between CNK and Linux also exist in BG/P communication libraries, but the program object files are the same.

A. Memory Benchmarks

We evaluated the memory performance using two benchmarks: a streaming copy benchmark and the random memory access benchmark from Section III.

The streaming copy benchmark is simple: it allocates a memory buffer, divides it in two, and copies data from one region to the other using PowerPC 450 parallel load/store instructions. The benchmark reports the theoretical peak memory performance if it is run in a noise-free environment. The results for several different buffer sizes can be found in Table III. With a 16 KB buffer size, which fits well in the L1 cache, all kernels perform equally well, because no TLB misses are caused by the benchmark even under standard Linux with 4 KB pages. With larger buffers, such as 4 MB (which fit in the L3 cache) and 256 MB (which do not fit), we see that memory performance using a standard Linux kernel is significantly lower than under CNK. Linux with Big Memory shows a 0.2% loss compared to CNK. This is a small performance hit, but it is still larger than expected, given that the noise level has been measured at just 0.03% (see Section III). We have not performed a detailed investigation of this phenomenon, but we suspect that instruction cache thrashing at context switch time caused by the OS tick update might be responsible; the benchmark issues a series of parallel load/store instructions, and the cost of cache thrashing might be higher than for an FPU benchmark.

Table III
Streaming copy benchmark (GB/s)

	16 KB	4 MB	256 MB
CNK	6.74	4.60	3.85
Linux 4 KB	6.73	3.85	3.35
Linux 64 KB	6.74	4.51	3.82
Linux Big Memory	6.74	4.59	3.84

In Section III, we showed the results of the random memory access benchmark on CNK and on Linux with paged memory; there was a large difference in performance. Table IV adds the result for Linux with Big Memory support; the gap to CNK is narrowed to 0.04%, or within the system noise level.

Table IV
Random memory access benchmark

	MB/s
CNK	44.70
Linux 4 KB	14.39
Linux 64 KB	16.40
Linux Big Memory	44.68

B. FFT Benchmark

To obtain more realistic performance numbers, we ran a simple FFT benchmark linked to the standard FFTW library (version 3.1.2). The benchmark uses a 2D array 5000 $\times$ 5000 in size, which consumes approx. 763 MB of memory. Table V shows the results of executing a forward FFT on the array. Clearly, standard Linux does not perform well; 64 KB pages improve performance by only 9%. On the other hand, Linux with Big Memory shows just a 0.08% performance loss compared to the CNK.

Table V
FFT benchmark

	Elapsed Time (s)
CNK	11.25
Linux 4 KB	21.43
Linux 64 KB	19.61
Linux Big Memory	11.24

C. NAS Parallel Benchmarks

So far we have shown that the Big Memory implementation definitely improves memory performance of applications on a single node. To evaluate parallel performance, we ran the NAS Parallel benchmarks [14] version 3.3 on both CNK and Linux with Big Memory support. We ran the experiment on 1024 nodes, in SMP mode (one process per

node), using class C problem sizes. The results are shown in Table VI. The performance is close: Linux runs were slower on most benchmarks compared to those using CNK by approx. 0.1% to 0.7%, with the exception of the IS benchmark, where Linux was 0.5% faster.

Although the table contains only the results using 1024 nodes, we have submitted most NAS Parallel benchmarks with 256, 512, and 1024 nodes to investigate scalability. We haven't observed any scaling issue on our Linux kernel up to 1024 nodes.

Table VI
NAS PARALLEL BENCHMARK

Type	CNK (Mop/s)	Linux (Mop/s)	Linux/CNK
IS	3991	4010	1.005
CG	15749	15707	0.997
MG	134955	134380	0.996
FT	96594	96385	0.998
LU	40890	40617	0.993
EP	2503	2500	0.999
SP	106009	105709	0.997
BT	165240	164777	0.997

D. LOFAR Online Central Processing

We had the opportunity to observe the performance of Big Memory in a real-life application. LOFAR is a radio telescope being built in the Netherlands [15]. In contrast to current radio telescopes that employ custom-built hardware as a correlator, LOFAR uses a Blue Gene/P supercomputer. Our previous work discusses the LOFAR central processor in more detail [7], [16].

LOFAR stations stream UDP/IP data directly into the Blue Gene/P I/O nodes at a rate of slightly more than 3 Gbps. These I/O nodes store the data in a main memory ring buffer, which is used to absorb network delays or temporary hiccups in the processing pipeline. From here the data is transported to the compute nodes, where the information is correlated.

Poor main memory performance of the I/O node running the default Linux kernel proved to be one of the major bottlenecks in trying to achieve optimum data throughput.

Table VII shows a breakdown of the tasks on the Blue Gene/P I/O nodes running the LOFAR online processing application in a fairly standard 16-bit observation mode and a more challenging 4-bit mode. The required CPU resources, in processor cores, are shown for the stock IBM I/O node kernel in the 16-bit mode and for a ZeptoOS compute node kernel, modified to run on the I/O nodes, for both the 16-bit and 4-bit modes. We see that using the original I/O node kernel would require almost two entire I/O node cores just to handle the ring buffer, and that there are not enough compute resources available to perform all the tasks in real time, even for the less demanding 16-bit mode.

We used a slightly modified ZeptoOS compute node kernel, including support for the Ethernet device and excluding compute node specific devices like the torus network, on the I/O node. We reserved 1536 MB of main memory as the Big Memory area and used it for the ring buffer using six 256 MB TLB entries. The I/O node application was also adapted to copy 128 UDP/IP packets into the ring buffer at once, instead of one at a time. These two optimizations reduce the resources required to copy data into the ring buffer by more than 600%, about half of which can be attributed to the lack of TLB misses in the Big Memory area.

The more challenging 4-bit mode significantly increases the potential number of TLB misses. Although limitations in the default kernel make direct comparison impossible, smallscale memory benchmarks show a more than 500% reduction in required CPU cycles using Big Memory when copying UDP/IP packets one by one.

Clearly, the access pattern of the LOFAR I/O node application is very susceptible to performance hits caused by TLB misses. With the stock I/O node kernel, the processor was unable to achieve our throughput requirements. Preventing TLB misses for at least part of the application, combined with several other optimizations not discussed here, allowed us to reduce CPU load considerably, increasing I/O node performance to well beyond our original requirements.

Table VII
LOFAR I/O NODE PROCESSING (#CORES)

	Stock	ZeptoOS	
	16 bit	16 bit	4 bit
Receive UDP/IP packets	1.44	1.44	1.22
Copy data to ring buffer	1.80	0.27	0.37
Send ring buffer to CN	1.40	0.52	0.61
Receive data from CN	1.00	0.10	0.35
Send results to storage	0.40	0.32	0.64
Total system load	151%	66.5%	79.7%

VI. CONCLUSIONS

This paper presented the implementation of Big Memory support for BG/P Linux—a transparent, flat memory space for computational processes. Big Memory addresses two major issues encountered when attempting to run high-performance code on the BG/P Linux: poor memory performance caused by TLB misses handled in software, and the difficulties of writing an efficient communication stack caused by the limitations of the BG/P torus' DMA engine.

Our experiments have shown that benchmarks running under a standard Linux kernel with paged memory can run up to three times slower than under CNK. With Big Memory support, Linux is slower by only 0.03%–0.2%. We think that the 0.03% loss is due to the time spent executing the OS tick interrupt handler; with some benchmarks this can grow to 0.2%, presumably because of instruction cache thrashing. With further kernel tuning, it should be possible to reduce the noise level to close to zero. Experiments at scale showed a slowdown of well under 1%.

Employing Big Memory on the I/O nodes was instrumental in reducing the I/O node CPU resources required for LOFAR online central processing. A 500–600% performance increase was observed in key parts of the application, allowing the I/O nodes to achieve their required throughput.

Our modifications to the Linux kernel are relatively small, principally because we focused exclusively on the requirements of computational processes, rather than trying to solve the problem in a generally applicable way, which would have been far more complicated. We maintain two versions of Linux kernel, and we found porting the Big Memory patches between the kernels to be straightforward.

Contemporary parallel architectures such as Blue Gene/P, Cray XT5, and Roadrunner use commodity CPUs such as Intel Xeon, AMD Opteron, or IBM PowerPC, instead of designs dedicated to computational environments. The memory management units in these processors are essentially designed to support a highly multitasking environment. It would be interesting if future designs had hardware support for computational process address space similar to the Big Memory area that we implemented, to allow for a seamless coexistence of high-performance applications and standard Unix processes on the compute nodes.

Along with the Big Memory implementation, we conceived the idea of a special process that the kernel treats differently from other processes. In the case of Big Memory, the kernel creates a different application address space, and we showed that this idea works for compute nodes. We have also experimented with other uses of this feature, such as disabling nonessential interrupts when a computational process gets scheduled in order to reduce the system noise.

Our current implementation is suitable for benchmarking and simple applications. We need to make several improvements in the quality of implementation, such as the granularity of the Big Memory area. So far, we have had little experience with running large, real-world applications on the compute nodes using our modified kernel; this area will be explored as part of our future research.

Acknowledgments: We thank IBM's Todd Inglett, Thomas Musta, Thomas Gooding, George Almási, Sameer Kumar, Michael Blocksome, and Robert Wisniewski for their advice on programming the Blue Gene hardware. We also thank our past summer interns Peter Boonstoppel, Hajime Fujita, Satya Popuri, and Taku Shimosawa, who contributed to the ZeptoOS kernel. Additionally, we thank Astron's John W. Romein, who evaluated Big Memory on the I/O nodes.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory.

REFERENCES

[1] "IBM Blue Gene," http://www.research.ibm.com/bluegene/.

[2] J. E. Moreira *et al.*, "Blue Gene/L programming and operating environment," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 367–376, Mar. 2005.

[3] J. E. Moreira *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.

[4] "ZeptoOS project," http://www.zeptoos.org/.

[5] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proceedings of the 8th IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sep. 2006.

[6] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, Mar. 2008.

[7] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman, "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008, pp. 153–162.

[8] IBM Blue Gene team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, pp. 199–220, 2008.

[9] K. Chen, R. Seth, and H. Nueckel, "Improving enterprise database performance on Intel Itanium architecture," in *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, Jul. 2003, pp. 98–108.

[10] E. Shmueli, G. Almási, J. Brunheroto, J. Castaños, G. Dózsa, S. Kumar, and D. Lieber, "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, Jul. 2008, pp. 165–174.

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

[12] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation*, ser. ACM SIGOPS Operating Systems Review, vol. 36, Boston, MA, Dec. 2002, pp. 89–104.

[13] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "Operating system issues for petascale systems," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 29–33, Apr. 2006.

[14] D. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[15] H. R. Butcher, "LOFAR: First of a new generation of radio telescopes," in *Proceedings of SPIE*, vol. 548, Oct. 2004, pp. 537–544.

[16] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart, "Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer," in *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, Cambridge, MA, Jul. 2006, pp. 59–66.