Performance and Scalability Evaluation of "Big Memory" on Blue Gene Linux *

Kazutomo Yoshii[1], Kamil Iskra[1], Harish Naik[1], Pete Beckman[1,2], P. Chris Broekema[3]

[1]Mathematics and Computer Science Division
[2]Leadership Computing Facility
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA

[3]ASTRON
Netherlands Institute for Radio Astronomy
Oude Hoogeveensedijk 4, Dwingeloo, The Netherlands

*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.

Proposed running head: **Big Memory on Blue Gene Linux**

Kazutomo Yoshii
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439, USA
phone: +1-630-252-4476
fax: +1-630-252-5986
email: kazutomo@mcs.anl.gov
(corresponding author)

Kamil Iskra
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439, USA
email: iskra@mcs.anl.gov
phone: +1-630-252-7197
fax: +1-630-252-5986

Harish Naik
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439, USA
email: hnaik@mcs.anl.gov
phone: +1-630-252-2029
fax: +1-630-252-5986

Pete Beckman
Leadership Computing Facility and
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439, USA
email: beckman@mcs.anl.gov
phone: +1-630-252-9020
fax: +1-630-252-5986

P. Chris Broekema
Netherlands Institute for Radio Astronomy
Oude Hoogeveensedijk 4,
Dwingeloo, The Netherlands
email: broekema@astron.nl
phone: +31-521-595-181
fax: +31-521-595-101

Abstract

We address memory performance issues observed in Blue Gene Linux and discuss the design and implementation of "Big Memory"—an alternative, transparent memory space introduced to eliminate the memory performance issues. We evaluate the performance of Big Memory using custom memory benchmarks, NAS Parallel Benchmarks, and the Parallel Ocean Program, at a scale of up to 4096 nodes. We find that Big Memory successfuly resolves the performance issues normally encountered in Blue Gene Linux. For the ocean simulation program, we even find that Linux with Big Memory provides better scalability than does the lightweight compute node kernel designed solely for high-performance applications. Originally intended exclusively for compute node tasks, our new memory subsystem dramatically improves the performance of certain I/O node applications as well. We demonstrate this performance using the central processor of the LOw Frequency ARray radio telescope as an example.

Keywords: Linux, Blue Gene, OS kernel, memory performance, TLB

1 Introduction

The Blue Gene architecture [9, 14], 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) [15] 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 importantly, 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 [20], 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 [2, 4] and on I/O forwarding [11]. 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 [10] introduced in 2007 to replace the original Blue Gene/L design [14].

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, 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 [11]. 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 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.

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 a cost of a few hundred CPU cycles (0.2 to 0.3 μs). Therefore, 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 ad-

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

vantage 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, on the other hand, 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.

2 Related Work

Linux does provide support for large memory pages, through the *hugetlbfs* [7] 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. [19] 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 [8], 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 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 on accessing floating-point constants.

Navarro et al. [16] 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 massively parallel machines.

3 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 [2, 4].

Figure 1 shows the results obtained by using our OS noise measurement benchmark Selfish [3] 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 with 4 kB pages and 100 Hz timer interrupt 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.

Table 1 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 $\times$ 850 MHz yields 3.4 Gflops). Linux is only 0.09% slower, which is higher than the noise level measured earlier, probably because the noise measurement experiment does not account for noise events smaller than 1 μs.

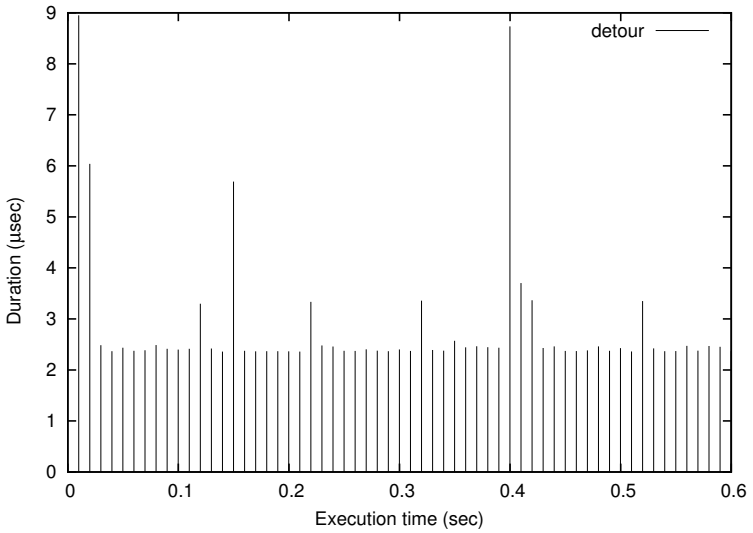

Figure 1: Noise on BG/P Linux

Table 1: CNK vs Linux: FPU benchmark

CNK (Gflops)	Linux (Gflops)	Performance Loss (%)
3.400	3.397	0.09

3.1 Memory Benchmarks

We evaluated memory performance using two benchmarks: a streaming copy benchmark and a random memory access benchmark. Under Linux, we tested using two different page sizes: 4 kB and 64 kB.

The streaming copy benchmark is simple: it allocates a memory buffer, divides it in two, and copies data from one region to the other. The results for several different buffer sizes can be found in Table 2. With a 16 kB buffer size, which fits well in the L1 cache, the Linux kernel incurs approx. 1.2% performance loss. This is higher than the overhead measured earlier; since no TLB misses occur during the execution of this benchmark, we suspect that Linux timer interrupts thrash the L1 cache. With a 256 kB buffer size, which is the maximum size that 4 kB TLB entries can cover (64 TLB entries $\times$ 4 kB), we observe that the performance under Linux with 4 kB pages drops. We do not see any significant performance drop at 4 MB, even though this is the maximum size that 64 kB TLB entries can cover. The reason is that the overhead of a TLB miss becomes very small: updating a TLB entry takes approx. 0.2 μs, while loading a 64 kB page takes approx. 60 μs, which yields a 0.3% performance loss. A TLB miss also thrashes several data cache lines, thus adding more overhead, so the performance loss we have observed is within our expectations.

Table 3 presents the results of our random memory access benchmark. This benchmark reads data from a

16 kB, 256 kB, 4 MB, or 16 MB memory region in a random fashion. The results show that the performance drops dramatically at 256 kB on Linux with 4 kB pages and at 4 MB on Linux with 64 kB pages. Many applications deal with buffers larger than 4 MB, so clearly, a more radical solution is needed than a mere increase of the page size by an order of magnitude.

Table 2: CNK vs Linux: Streaming Copy Benchmark

Size	CNK (MB/s) (σ)	Linux 4 kB (MB/s) (σ)	Linux 64 kB (MB/s) (σ)	Performance Loss 4 kB (%)	Performance Loss 64 kB (%)
16 kB	2158.00 (0.03)	2131.67 (0.30)	2130.81 (0.40)	1.22	1.26
256 kB	1037.13 (0.03)	993.04 (6.84)	1031.44 (0.18)	4.25	0.55
4 MB	1037.39 (0.00)	993.51 (1.81)	1028.49 (2.50)	4.23	0.86
16 MB	1037.38 (0.02)	992.96 (1.73)	1028.08 (2.00)	4.28	0.90

Table 3: CNK vs Linux: Random Memory Acccess Benchmark

Size	CNK (MB/s) (σ)	Linux 4 kB (MB/s) (σ)	Linux 64 kB (MB/s) (σ)	Performance Loss 4 kB (%)	Performance Loss 64 kB (%)
16 kB	810.29 (0.00)	807.34 (0.23)	807.90 (0.08)	0.36	0.29
256 kB	810.36 (0.00)	67.16 (0.00)	810.16 (0.03)	91.71	0.02
4 MB	186.47 (0.00)	22.11 (0.13)	54.23 (0.02)	88.14	70.92
16 MB	62.60 (0.00)	17.76 (0.00)	22.66 (0.00)	71.63	63.80

3.2 NAS Parallel Benchmarks – Serial

To study the single node performance under different loads, we ran NAS Parallel Benchmarks (NPB) [1] version 3.3 in serial mode using the class A problem size. The executables were compiled using the IBM XL compiler and we could use the same binaries for both CNK and Linux. The results are in Figure 2.

The IS result shows a large performance gap between CNK and Linux. This benchmark performs random memory accesses on a larger buffer, so the performance loss is close to the numbers from Table 3. Surprisingly, the CG benchmark on Linux with 64 kB pages runs faster than on CNK. We found that CNK enables the L2 optimistic prefetch, while Linux does not. In this particular case, the prefetch results in a performance loss. Brunheroto et al. [5] describe the L2 optimistic prefetch on Blue Gene/L and they confirm that the prefetch negatively affects the CG, IS, and SP benchmarks.

3.3 System Calls

Table 4 shows a comparison of gettimeofday() performance. This is one of the benchmarks where Linux outperforms CNK. In this experiment gettimeofday() is a system call on both CNK and BG/P Linux; we

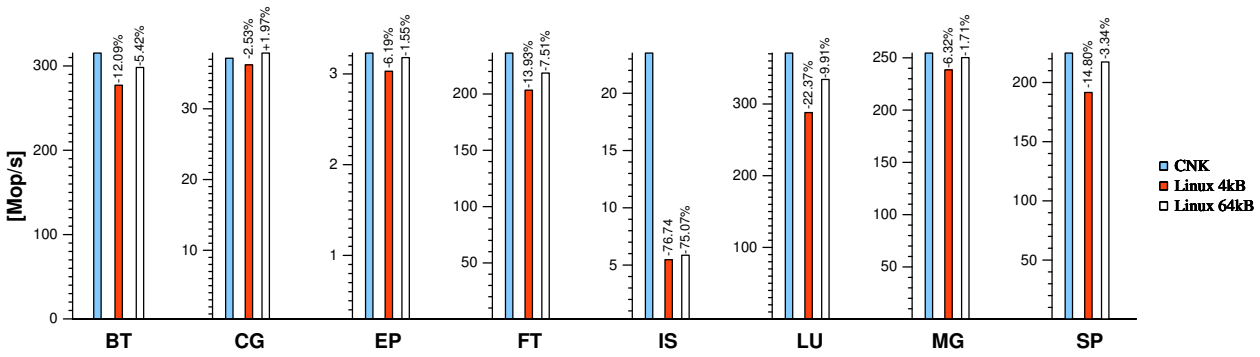

Figure 2: CNK vs Linux: NAS Parallel Benchmarks – Serial

don't use a user space gettimeofday(). The cost of a gettimeofday() system call on CNK is almost eight times higher than that on Linux.

We found that at least two factors contribute to this difference. First, CNK saves all possible registers on system call entry, while Linux saves a minimum set. Additional register saves and restores not only take time but also thrash the L1 cache; the impact on performance depends on how cache-intensive the application is. Second, the gettimeofday() system call on CNK requires multiple, expensive 64-bit integer operations to convert the CPU time stamp counter to the wall-clock time; under Linux, 64-bit operations are not needed because the Linux kernel maintains the wall-clock time using the timer interrupt, so calculating the time difference from the last tick update using 32-bit operations is sufficient. Even if we implement gettimeofday() in user space on CNK, it costs approx. 230 CPU cycles (0.27 μs), although reading a 64-bit time stamp counter itself costs only about 10 CPU cycles (0.01 μs).

Table 4: CNK vs Linux: System Call Performance

	CNK (μs/call)	Linux (μs/call)	Performance Loss (%)
`gettimeofday`	3.91	0.51	**−86.96**

4 Big Memory

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

4.1 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 3 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.

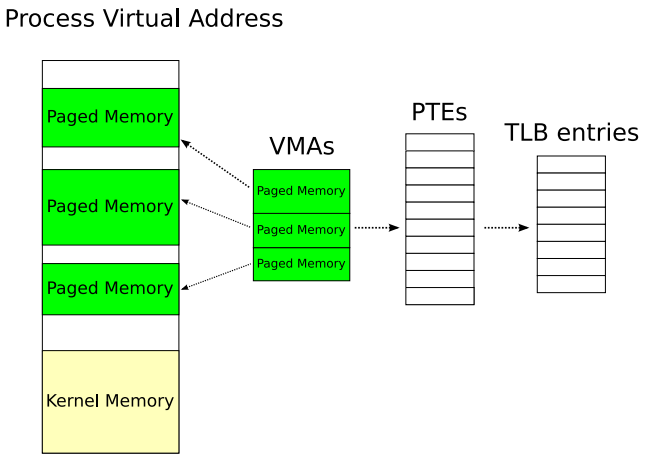

Figure 3: Memory Management in Linux

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. 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 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 TLB entries to cover kernel memory space, so TLB entries associated with kernel memory are not flushed when switching context.

4.2 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 2, 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 applications. Our approach is to allow only a single computational process to run at a time, to have 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 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 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.

4.3 ELF Binary Interpreter

To load a ZCB into Big Memory, we have modified the Linux kernel ELF binary interpreter, specifically, 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.

4.4 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 and text). 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 4). 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.

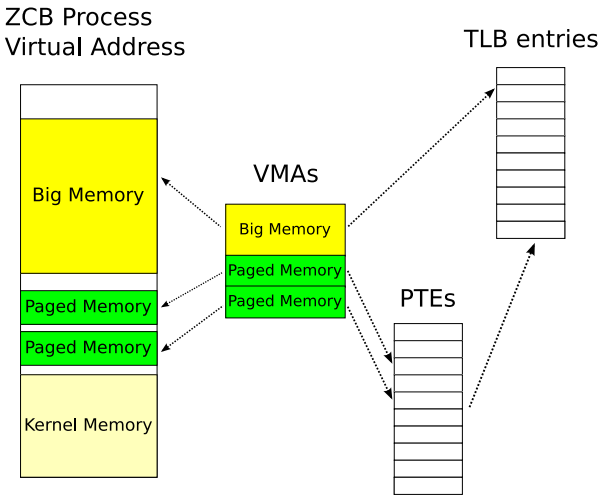

Figure 4: ZCB Process Address Space

4.5 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 5). 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.

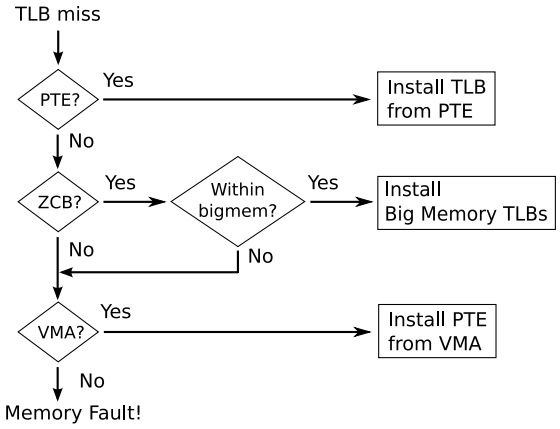

Figure 5: Big Memory Fault-Handling Flow

As shown in Figure 6, our Linux kernel partitions TLB entries in four groups: kernel mapping, paged memory, Big Memory, and device mappings. The last two groups are 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. The area size can be specified as a kernel parameter. Three TLB 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.

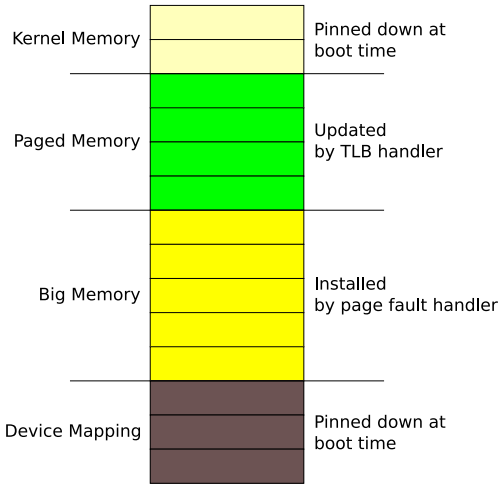

Figure 6: TLB Partitioning in the Zepto Kernel

4.6 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 a regular Linux process; however, it cannot call `fork` itself, since the fork code 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 the context of another process. 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 such as *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. The `mprotect` system call is not implemented, so debugging mechanisms that depend on `mprotect` will not work.

5 Performance Evaluation

To evaluate single-node performance on Linux with Big Memory, we ran our memory microbenchmarks and the serial implementation of the NAS Parallel Benchmarks. For parallel performance evaluation, we used (the parallel implementation of) NPB and the Parallel Ocean Program (POP). The experiments were performed on BG/P compute nodes with three different kernels: IBM CNK, Linux with 64 kB pages, and our enhanced Linux with Big Memory support. We used the IBM XL compiler that supports PowerPC 450 instructions to compile the benchmarks.

Sequential binaries are compatible between the kernels. Parallel binaries, however, can be run only on CNK and Linux with Big Memory because we cannot run MPI applications under Linux without Big Memory due to limitations of IBM's low-level communication library. To make the comparison as fair as possible, we rebuilt communication libraries for CNK and Linux from the same sources: BG/P-patched MPICH version 1.0.7 and Deep Computing Messaging Framework library (DCMF) version 1.0. Differences between the kernels result in slightly different binaries of the communication libraries and consequently in different parallel binaries, but the intermediate application object files were the same.

5.1 Memory Benchmarks

In Section 3.1, we showed the results of both the streaming copy benchmark and the random memory access benchmark; Linux had serious performance issues with random memory access.

We ran the memory benchmarks on Linux with Big Memory ("Bigmem"). The results are in Tables 5 and 6; they show that Big Memory completely eliminates the performance problem of random access to paged memory. Big Memory also slightly improves the streaming access pattern. Interestingly, we observe that with 16 kB and 256 kB buffer sizes the performance loss with Big Memory is slightly larger than with 64 kB pages. We do not see any significant numbers of TLB misses in either case; instead, the slowdown is caused by involuntary context switches. To keep Big Memory mapping private to the computational process, the associated TLB entries are flushed and re-installed when the process is scheduled in and out. The cost of removing Big Memory TLB entries is slightly higher than the cost of a single TLB miss. We also note that the performance loss with the streaming access is proportional to the memory bandwidth: 0.6% performance loss per GB/s.

Table 5: CNK vs Linux: Streaming Copy Benchmark

Size	CNK (MB/s) (σ)	Linux 64 kB (MB/s) (σ)	Linux Bigmem (MB/s) (σ)	Performance Loss 64 kB (%)	Performance Loss Bigmem (%)
16 kB	2158.00 (0.03)	2130.81 (0.40)	2130.15 (0.42)	1.26	1.29
256 kB	1037.13 (0.03)	1031.44 (0.18)	1030.85 (0.06)	0.55	0.61
4 MB	1037.39 (0.00)	1028.49 (2.50)	1031.67 (1.52)	0.86	0.55
16 MB	1037.38 (0.02)	1028.08 (2.00)	1031.87 (2.03)	0.90	0.53

Table 6: CNK vs Linux: Random Memory Acccess Benchmark

Size	CNK (MB/s) (σ)	Linux 64 kB (MB/s) (σ)	Linux Bigmem (MB/s) (σ)	Performance Loss 64 kB (%)	Performance Loss Bigmem (%)
16 kB	810.29 (0.00)	807.90 (0.08)	807.04 (0.06)	0.29	0.40
256 kB	810.36 (0.00)	810.16 (0.03)	809.51 (0.47)	0.02	0.10
4 MB	186.47 (0.00)	54.23 (0.02)	186.18 (0.01)	70.92	0.16
16 MB	62.60 (0.00)	22.66 (0.00)	62.52 (0.00)	63.80	0.13

5.2 NAS Parallel Benchmarks – Serial

In Section 3.2, we showed the results of experiments with the serial implementation of NPB on both CNK and paged Linux.

Figure 7 adds the results for Linux with Big Memory support. The results show that Big Memory clearly improves the memory performance, e.g., the huge loss observed with the IS benchmark is gone. On the other hand, the performance of the CG benchmark is now in line with the CNK, because unlike paged memory, Big Memory uses the same optimistic prefetch configuration as the CNK. The SP benchmark performance improves slightly with Big Memory Linux—we suspect the IBM XL compiler is responsible; the particular binary that we compiled somehow negatively affected the performance only on CNK.

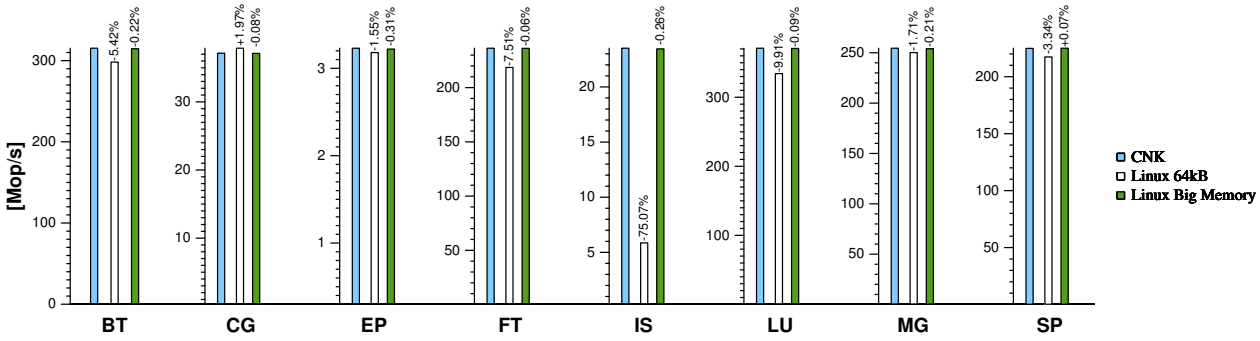

Figure 7: CNK vs Linux: NAS Parallel Benchmarks – Serial

5.3 NAS Parallel Benchmarks – Parallel

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 NPB in parallel mode on both CNK and Linux with Big Memory support.

We ran all the NPB benchmarks on both 1024 and 4096 nodes (except IS, which supports at most 1024 nodes), in SMP mode (one process per node), using the class D problem size.

The results are shown in Figure 8. The performance is very close: Linux runs were slower in all cases by 0.1–1.3%. The performance loss with the LU and CG benchmark increases at 4096 nodes, while the loss with the MG, FT, SP, and BT benchmarks decreases. We have not yet done a detailed investigation of these trends. We suspect that for benchmarks that improve as we scale up, the system noise might be hidden by the hardware-assisted collective operations and DMA transfers (which are not affected by noise) or by imperfectly balanced computations that force the majority of processes to wait doing nothing anyway. The performance loss with the EP benchmark stays pretty much independent of the number of nodes, since the benchmark is an embarrassingly parallel random-number generator. The 1.3% performance loss observed is consistent with the loss we saw earlier with the streaming copy benchmark (see Table 5).

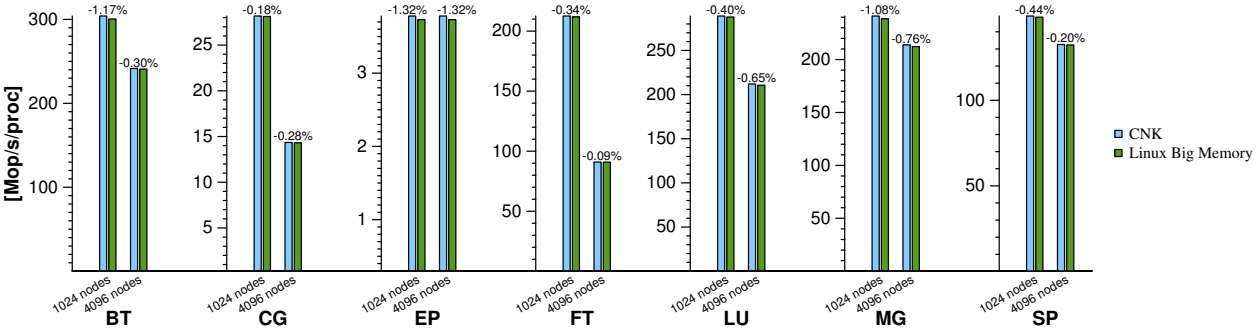

Figure 8: CNK vs Linux: NAS Parallel Benchmarks – Parallel

5.4 Parallel Ocean Program

In addition to evaluating the performance using synthetic benchmarks, we wanted to confirm the results using a real-world application. We chose the Parallel Ocean Program (POP) [12, 13], a wellknown parallel application for studying the ocean climate system. POP is notorious for being sensitive to noise, so it seemed like an excellent candidate.

We used unmodified POP version 2.0.1 from the POP website and the X1 benchmark data set. We

ran it at the scale of 64 to 4096 nodes, in SMP mode. To be able to run using different node counts, we adjusted the `block_size_x` and `block_size_y` parameters, keeping both `max_blocks_clinic` and `max_blocks_tropic` at 1.

The results are shown in Figure 9. Somewhat surprising is the fact that Linux runs scale better than CNK. We found that POP calls `gettimeofday` about 100,000 times on each node, irrespective of the total number of nodes, resulting in a constant 0.36 second penalty on CNK due to the higher system call overhead discussed in Section 3.3. As the run times decrease with increasing node count, this overhead becomes ever more significant, eventually allowing Linux to overtake CNK if users are not aware of the high cost of `gettimeofday` on CNK. To be fair in terms of memory subsystem comparison, we replaced the `gettimeofday()` system call with a user space implementation and conducted an experiment on 4096 nodes. In this case Linux is 1.01% slower than CNK, which is less than shown, for example, in the NAS EP benchmark on 4096 nodes. This result indicates that CNK system calls not only are slow but also thrash the cache, since the absolute difference in run times under CNK using the two timer implementations, which was 0.68 second, is almost twice the cost of the system calls measured using a synthetic benchmark.

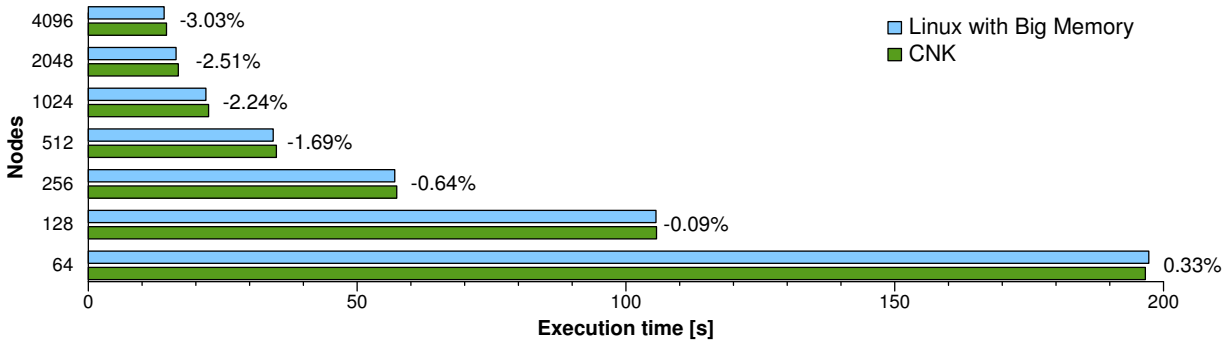

Figure 9: CNK vs Linux: POP Execution Time

5.5 LOFAR Online Central Processing

We've shown in the previous sections that Big Memory performs well in several synthetic benchmarks on the compute nodes. Our earlier work showed that, for some highly I/O bound applications, it makes sense to run *application software* on the I/O nodes [11]. Since I/O nodes run Linux, these applications may similarly benefit from Big Memory.

The LOw Frequency ARay(LOFAR) radio telescope is being built in the Netherlands [6]. In contrast to current radio telescopes that employ custom-built hardware correlators, LOFAR uses a Blue Gene/P

supercomputer. The LOFAR central processor is discussed in more detail elsewhere [18]; here we focus on the performance of the real-time application running on the Blue Gene/P I/O nodes.

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 all station pairs are correlated. Correlated data is transported back to the I/O nodes, which forward it to a storage cluster for further processing. Poor main memory performance of the I/O node running the unoptimized Blue Gene/P Linux kernel proved to be a major I/O bottleneck.

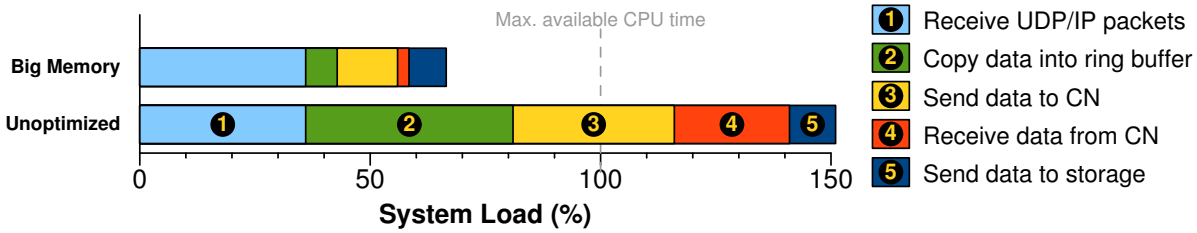

Figure 10: Blue Gene/P I/O Node Tasks in LOFAR Online Processing

Figure 10 shows a breakdown of the tasks on the Blue Gene/P I/O nodes running the LOFAR online processing application. We compare the performance of LOFAR's main observation mode using a highly optimized I/O node with Big Memory to the same observation mode running on an unoptimized I/O node.

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 for the ring buffer using six 256 MB TLB entries. The I/O node application was also adapted to copy 128 bytes into the ring buffer at once, instead of one at a time, using Blue Gene specific instructions. These two optimizations reduce the resources required to copy data into the ring buffer by some 500%. A small memory benchmark that uses the same access pattern, but without the optimizations mentioned before, proves that half of this gain can be directly attributed to Big Memory(see Table 7).

Table 7: I/O Node Memcpy Performance

Unoptimized (MB/s)	Big Memory (MB/s)	Performance increase (%)
212	505	251

The access pattern of the LOFAR I/O node application is susceptible to performance hits caused by

TLB misses. Data must be read in such a way that a TLB miss is almost guaranteed. With the stock I/O node kernel, the I/O node processor was unable to achieve our throughput requirements. Preventing TLB misses for at least part of the application, combined with a low-overhead, high-performance I/O node to compute node protocol [17], allowed us to reduce CPU load considerably, increasing I/O node performance to well beyond our original requirements.

6 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 OS noise measurement benchmark shows that Linux kernel with 4 kB pages and 100 Hz uses only 0.027% of the CPU time. The FPU benchmark ,which is a tight loop of FMA instruction, suffers 0.09% of the performance loss compared to CNK. We think that the 0.09% loss is due to the time spent executing the OS tick interrupt handler.

Our experiments have shown that memory benchmarks running under a standard Linux kernel with 4 kB pages suffer up to 92% of performance loss compared to CNK. Increasing the Linux kernel page size to 64 kB narrows the gap; the worst case performance loss drops to 71%. Once we introduce Big Memory support, the worst case performance loss drops to just 1.3%.

We ran NAS Parallel Benchmarks at up to 4096 nodes and confirmed that Linux has no problems at this scale. In fact, with the majority of the benchmarks, the performance loss decreases with an increasing total number of nodes. We also ran the Parallel Ocean Program (POP) and the results show that for this application Linux scales better than does CNK. We found that the high overhead of system calls under CNK is responsible.

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% 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.

As of this writing, many contemporary parallel architectures use commodity CPUs instead of designs dedicated to computational environments. IBM Blue Gene/P uses IBM PowerPCs, Cray XT5 uses AMD Opterons, and IBM Roadrunner uses both the Opterons and IBM Cells that contain a PowerPC and special-purpose cores, for example. 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 came up with 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 in order to reduce system noise when a computational process gets scheduled in.

Our current implementation is suitable for benchmarking and simple applications. As of this writing, only one MPI rank per node is supported; applications have to use threads in order to exploit all four cores. Internally we have already extended Big Memory to support four processes per node, but the communication stack is not ready yet for this mode at this point; we will eliminate this limitation as soon as possible. We also need to make several improvements in the quality of implementation, such as the granularity of the Big Memory area. So far, we have not observed any parallel performance degradation caused by OS noise up to 4096 nodes. Confirming this situation at very large scales and investigating why OS noise does not seem to affect the HPC applications will be 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] D. Bailey et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[2] 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, Sept. 2006.

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

[4] 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*, 11(1):3–16, Mar. 2008.

[5] J. R. Brunheroto, V. Salapura, F. F. Redigolo, D. Hoenicke, and A. Gara. Data cache prefetching design space exploration for Blue Gene/L supercomputer. In *Proceedings of the 17th IEEE International Symposium on Computer Architecture on High Performance Computing*, pages 201–208, Rio de Janeiro, Brazil, Oct. 2005.

[6] H. R. Butcher. LOFAR: First of a new generation of radio telescopes. In *Proceedings of SPIE*, volume 548, pages 537–544, Oct. 2004.

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

[8] D. Gibson and A. Litke. libhugetlbfs. http://sourceforge.net/projects/libhugetlbfs.

[9] IBM Blue Gene. http://www.research.ibm.com/bluegene/.

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

[11] 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*, pages 153–162, Salt Lake City, UT, Feb. 2008.

[12] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP): Research articles. *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[13] D. J. Kerbyson and P. W. Jones. A performance model of the Parallel Ocean Program. *International Journal of High Performance Computing Applications*, 19(3):261–276, 2005.

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

[15] 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.

[16] 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*, volume 36 of *ACM SIGOPS Operating Systems Review*, pages 89–104, Boston, MA, Dec. 2002.

[17] J. W. Romein. FCNP: Fast I/O on the Blue Gene/P. In *Parallel and Distributed Processing Techniques and Applications (PDPTA'09)*, volume 1, pages 225–231, Las Vegas, NV, July 2009.

[18] J. W. Romein, P. C. Broekema, J. D. Mol, and R. V. van Nieuwpoort. The LOFAR Correlator: Implementation and Performance Analysis. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, Bangalore, India, January 2010.

[19] 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*, pages 165–174, Kos, Greece, July 2008.

[20] ZeptoOS project. http://www.zeptoos.org/.