Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK

Mark Giampapa* Thomas Gooding+
Todd Inglett+ Robert W. Wisniewski*

***Abstract*—The Petascale era has recently been ushered in and many researchers have already turned their attention to the challenges of exascale computing[2]. To achieve petascale computing two broad approaches for kernels were taken, a lightweight approach embodied by IBM Blue Gene's CNK, and a more fullweight approach embodied by Cray's CNL. There are strengths and weaknesses to each approach. Examining the current generation can provide insight as to what mechanisms may be needed for the exascale generation. The contributions of this paper are the experiences we had with CNK on Blue Gene/P.**

We demonstrate it is possible to implement a small lightweight kernel that scales well but still provides a Linux environment and functionality desired by HPC programmers. Such an approach provides the values of reproducibility, low noise, high and stable performance, reliability, and ease of effectively exploiting unique hardware features. We describe the strengths and weaknesses of this approach.

***Index Terms*—Lightweight Kernel, Supercomputer, HPC system**

I. Introduction

Approaches to HPC operating system kernels differ in the amount of functionality provided by the kernel. Kernels that provide less than full functionality are typically referred to as lightweight. Traditional wisdom viewed such kernels as lacking significant function and providing an unfamiliar programming environment. Supporting this notion were previous generation kernels such as Catamount[19] (from Sandia, UNM, and Cray) for Cray supercomputers and IBM's CNK for Blue Gene/L[14], [23].

There are two approaches to get software to scale well to large supercomputers. It is possible to start with an existing software base, analyze its scalability bottlenecks, fix them, and iterate. Alternatively, a new code base can be designed from the beginning to scale to large numbers of nodes. This was the approach for the kernel on Blue Gene/L and Catamount. The current generation instantiations of Blue Gene/P's CNK[17], CNW[12], and Kitten[3] and Palacios[22], [21] maintain the scalability philosophy, but provide more functionality and are dispelling the notion that a lightweight kernel has to be lightweight on functionality.

In June 2007 Blue Gene/P became available. The Compute Node Kernel (CNK) running on BG/P is consistent in design philosophy with BG/L's kernel. It is lightweight, very low noise, designed to scale to hundreds of thousands of nodes, and continues to provide performance reproducibility. However, on BG/P we leveraged open source software to provide a large range of POSIX semantics and a familiar Linux programming environment. In addition to leveraging open source software, much of the software stack on BG/P is itself open source, see Figure 1.

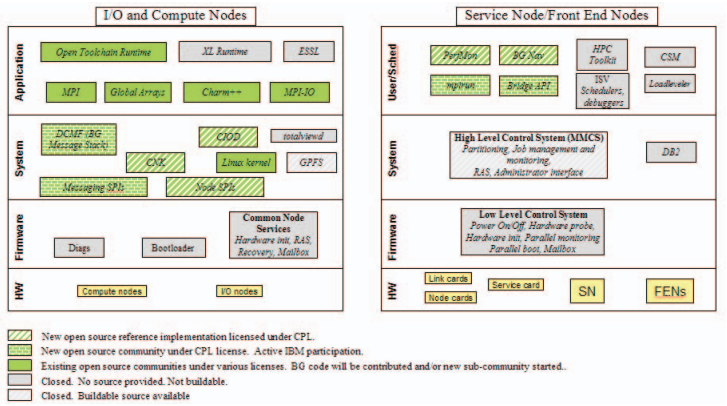

Fig. 1. Blue Gene/P Open Source Software Stack

Concerns expressed about the Light-Weight Kernel (LWK) approach tend to focus on the [lack of] completeness of a Linux environment. There are different reasons why a Linux environment is viewed positively. Ordered by decreasing requirements these include: 1) desire to run standard applications out-of-the-box, 2) ability to work in an open source environment and leverage the pool of people familiar with the Linux model, 3) desire for a familiar runtime environment, pthreads, glibc, etc., 4) ability to make standard Posix system calls. We will denote providing a standard Linux code base running on a compute node with all functionality allowing applications to run out-of-the-box, as a Full-Weight Kernel (FWK) approach. A common distinguishing characteristic of LWKs it that they set up resources that are then passed to the application to use directly, while FWKs tend to maintain ownership.

Although Linux provides advantages, from a research perspective it does not allow as much flexibility or ease of exploration. Thus, for researchers wanting to explore the effects of different kernel policies on HPC applications, CNK or Kitten provides a more easily modifiable base. Further, the Linux community has not eagerly embraced HPC patches, while they fit well into CNK or Kitten. The downside of exploring HPC innovations in an LWK is that all applications may not run out-of-the-box and thus it may be more difficult understanding the benefits of an idea on a particular application of interest. In addition to the research advantages of being unencumbered by the Linux model and weight, an LWK approach offers other advantages:

1) High performance: the focus of an LWK is on perfor-

*IBM T. J. Watson Research Center. "The Blue Gene/P project has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the U.S. Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331."

+IBM Rochester

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

mance. System design involves tradeoffs. LWKs tend to choose the high performance path, while FWKs tend to focus on generality

2) Low noise and performance stability: an LWK approach provides a low noise environment with stable application performance facilitating optimization.
3) Ability to work with partial or broken hardware: bringing up a new chip is challenging. For supercomputer vendors who innovate with hardware, a simple small kernel to bring up a new chip offers a more controllable environment, allowing quick work-arounds to hardware bugs, and facilitates testing.
4) Reproducibility: related to the previous item, CNK provides a cycle-by-cycle reproducible environment critical to debugging within a chip and diagnosing problems across 100,000s of nodes.
5) Ability to customize to unique hardware features and system software requirements: An LWK is nimble, allowing it to be easily modified to leverage unique hardware for HPC applications, and to provide customized features for the system software to increase performance.

We chose to continue the LWK approach on Blue Gene/P. We do, however, recognize the value of having a Linux-like environment. By leveraging open source software components, CNK on BG/P provides a Linux-like environment while maintaining the above LWK advantages.

There are advantages and disadvantages to both LWKs and FWKs. The goals and contributions of this paper are not to argue that the community should pick one over the other, but to

- 1a: describe the benefits new chip bringup derives from a self-designed and fully-controllable LWK that have not been previously documented,
- 1b: describe design features of the BG/P CNK that were not included in the BG/L CNK[23]
- 2a: detail the advantages of using an LWK that would be difficult to achieve on an FWK Linux code base, and
- 2b: detail the advantages of using an FWK that are difficult to achieve with an LWK approach, and
- 3: dispel the notion that an LWK of necessity lacks Linux functionality, and
- 4: describe how we will extend CNK for next-generation machines with higher core and thread counts.

The rest of the paper is structured as follows. We start by setting our work in context by describing related work in Section II. In Section III we address contribution 1a by describing the technology we employed to help early debugging and reproducibility. Section IV addresses contributions 1b and 3 and describes how we leverage open source software and provide a Linux-like environment with CNK. Although a design and experiences paper, we present results in Section V from two perspectives: a description of the broad range of applications that can run on CNK to address contribution 3, and we show the low noise and performance benefits both in terms of absolute and stable performance to address 2a. We address contribution 2b in Sections VI and VII where we describe what from a standard Linux is not available under CNK, and We address 4 in Section VIII where we discuss CNK's extended thread affinity model and the challenge it presents in maintaining the LWK design philosophy. We conclude in Section IX.

II. Related Work

The two important aspects of our work are providing a lightweight design focusing on low noise and ability to customize to specific hardware and HPC applications' needs, and providing a familiar Linux programming environment for the application. Work focusing on a lightweight design at Sandia and the University of New Mexico has been ongoing for well over a decade[25]. That collaboration produced operating systems for several top computer systems including the Catamount[19] operating system that is part of the Red Storm[6] project. Catamount was designed to be lightweight to facilitate scalability. Its strategy of providing only the features necessary to support the application, and then giving control of the processor and hardware to the application, is consistent with CNK's implementation and yields a very low-noise kernel. On BG/P we enriched the operating environment, providing more flexibility and functionality, by leveraging and integrating Linux code e.g., glibc, NPTL. On-going work on Kitten[3] in combination with Palacios[22], [21] has similar goals of providing a much richer environment that more easily supports existing codes.

Another approach, taken by ZeptoOS[10], is to start with a Linux image and make modifications such as reserving memory during boot before Linux accesses it, and removing auxiliary daemons to reduce the noise. By doing so, ZeptoOS can study operating system issues for petascale architectures with 10K to 1M CPUs. A similar study undertaken by Shmueli et al.[26] looked at the issues that caused Linux not to scale to 10s of 1000s of nodes. They too observed that by modifying the memory management system of Linux, reducing the number of daemons, and synchronizing the remaining daemons, it was possible to get Linux to scale. Both these approaches inherit the structure and algorithms of Linux, and though they can be modified, Linux is a moving target and is not focused on the high-end HPC space. These types of solutions offer a viable approach to meet the needs of many HPC applications. We faced the additional challenge of providing a cycle-reproducible environment for chip bringup capable of running on partial or broken hardware. This would be challenging with Linux.

Other groups have taken the approach of providing a full-feature Linux allowing programs to run out-of-the-box. Both SGI's Altix[7] ICE Pleiades SUSE-based Linux machine installed at NASA[5], and Cray's CNL[18], [27] on Jaguar take this approach. The goal behind a full Compute-Node Linux (CNL) approach is to provide the rich set of operating system services and system calls that users and developers expect, and that their applications may require. These groups provide this functionality by reducing daemons and working on memory allocation issues, but try to upstream their work so the end system can be built by config options. The Linux community is not always receptive to all the changes needed, so

sometimes this approach also requires additional patches. This approach also tends not to be as performance reproducible as a lightweight approach demonstrated by application groups[11]. As we discussed in the introduction, each approach is a viable alternative and neither is necessarily superior to the other. Exploration along both paths is healthy for the community as learning from the other approach strengthens each.

Although not specifically targeted for supercomputers, the L4 Microkernel[4] provides an efficient lightweight microkernel on which addition operating service can be constructed. The microkernel itself shares many aims with an LWK approach. Another related project has been to get Plan 9 running on supercomputers[16]. This approach exposes the power of a supercomputer to applications allowing user-mode and kernel-mode components to be interchanged, and with services provided by these components able to be used by the application transparently whether the components are local or remote. The approaches provide different and potentially useful design points. As the HPC community progresses from petascale to exascale, it is likely combinations of aspects from many of these approaches will be useful.

III. Using CNK for SOC chip design and bringup

In this section we describe experiences we had using CNK as an active participant in the design of our BG/P chip and our experience with the critical role it played during chip bringup. While this primarily addresses contribution 1a, it also implicitly provides support for 2a.

The Blue Gene family of supercomputers utilizes System-On-a-Chip (SOC) technology. SOC has reliability advantages, and enables performance, cost, and packaging optimizations, but comes with some challenges. We found it valuable to employ a lightweight kernel approach so that CNK could participate actively in the Blue Gene/P chip architecture, design and analysis process, and the logic verification and debug process. To be engaged in these early stages, CNK was designed to be functional without requiring the entire chip logic to be working. The startup and runtime configuration of CNK contains independent control flags and configuration parameters that support it running even when many features of the BG/P hardware did not exist (during design) or were broken (during chip bringup). Computational kernels of important applications were extracted and executed in simulation, enabling performance measurements on a wide range of configurations even before major units of the chip were in place. As an example, the BG/P memory system contains L2 Cache configuration parameters that control the mapping of physical memory to cache controllers and to memory banks within the cache. CNK enabled application kernels to be run with varied mappings of code and data memory traffic to the L2 cache banks, allowing measurement of cache effects, and optimizing the memory system hierarchy to minimize conflicts. Sensitivities of applications to cache sizes and prefetching algorithms were measured. Using these controls also enabled verification of the logic, and measurement of performance, in the presence of artificially created conflicts.

A key feature of CNK is the ability to provide perfect, cycle by cycle, reproducibility on hardware of test cases and application codes. The CNK kernel low-core leverages aspects of the Blue Gene/L Advanced Diagnostic Environment[14]. Reproducibility enables debugging the hardware via logic scans, which are destructive to the chip state. This technique requires performing logic scans on successive runs, each scan taken one cycle later than on the previous run. The scans are assembled into a logic waveform display that spans hundreds or thousands of cycles, enabling logic designers to see events leading up to the problem under investigation.

CNK support for reproducibility requires that the kernel be able to tolerate a chip reset at any point, and be able to restart identically from that state each time. The only persistent state that exists in a BG/P chip is located in DRAM during Self-Refresh. CNK prepares for full reset by performing a barrier over all cores, rendezvousing all cores in the Boot SRAM, flushing all levels of cache to DDR, placing the DDR in self-refresh, and finally toggling reset to all functional units, which causes the chip to boot. Upon boot, CNK checks if it has been restarted in reproducible mode, and if so, rather than interacting with the service node, initializes all functional units on the chip and takes the DDR out of self-refresh. Following those steps, CNK runs through its initial startup sequence to reinitialize all critical memory contents. The number of cycles that CNK can run in reproducible mode is bounded only by the point at which external I/O becomes necessarily.

BG/P provides “Clock Stop” hardware that assists the kernel in stopping on specific cycles. A challenge with this hardware is that it only supports a single chip, and thus debugging a communication bug that spans multiple chips is difficult. To overcome this problem, the CNK reboot process was modified to use the BG/P Global Barrier network to coordinate reboots across multiple chips. Across these reboots intended to be multichip reproducible, the barrier network was set to remain active and configured, but special code ensured a consistent state in all arbiters and state machines involved in the barrier network hardware. This allowed one chip to initiate a packet transfer on exactly the same cycle relative to the other chip that was used to capture logic scans.

There are many bugs encountered when bringing up a new chip. As an example, we describe a bug where the above capabilities proved immensely helpful. In one area of the chip there was a borderline timing bug whose manifestation was dependent both on manufacturing variability and on local temperature variations or electrical noise during execution. The bug thus did not occur on every chip, nor did it occur on every run on a chip that had the potential to exhibit the problem. Consistent re-creation of the problem therefore proved elusive, and its nature prevented recreating it in simulation. One piece of evidence that lead to the bug being tracked down was wave forms (hardware traces) gathered on reproducible runs across multiple chips, and using those to determine characteristics of a chip at the point it diverged from the expected cycle-reproducible run.

Another important aspect of a lightweight approach manifests itself during chip design. During chip design the VHDL cycle-accurate simulator runs at 10HZ. In such an environment, CNK boots in a couple of hours, while Linux takes weeks. Even stripped down, Linux takes days to boot, making

it difficult to run verification tests. The examples in this section illustrate the advantages we garnered with CNK's reproducibility; these would be more difficult to achieve with non-lightweight approaches.

IV. The CNK Linux Environment

As described in the introduction, keeping CNK small and lightweight offers advantages. However, a trend in HPC applications over the last decade has been towards more complex applications requiring more functionality from the operating system environment. As Linux has gained broader acceptance in the HPC space, more applications are written assuming a Linux environment. To combine the objectives of a lightweight kernel, more functionality, and a Linux environment, we leveraged components from the open source community such as glibc, pthreads, etc., and layered them on top of CNK. Although work was needed to provide this integration, it was not significantly more than providing our own proprietary limited-functionality libraries, threading, etc. Once we provided the base layer of support for Linux packages in CNK, additional Linux functionality was also available. A strategy that leverages Linux is affected when modifications are made in the code base. The one advantage of drawing the line between glibc and the kernel, is that that interface tends to be more stable, while internal kernel interfaces tend to be more fluid. In this section we describe the three key areas of support, namely file I/O, runtime environment, including threading and dynamic linking, and memory management that are part of CNK's base layer of support.

A. File I/O

In previous work[23] we described how we achieved I/O scalability on BG/L. The strategy we took on BG/P reduced the complexity of maintaining POSIX semantics and increased the performance and scalability of I/O. A key difference from BG/L is that on BG/P each MPI process has a dedicated I/O proxy process, and each thread within the MPI process has a dedicated thread within the I/O proxy process. As with BG/L, the offload strategy performs aggregation allowing a manageable number of filesystem clients, and reduces the noise on the compute nodes. In this section, we describe the details of how I/O offload works on BG/P

When an application makes a system call that performs I/O, CNK marshals the parameters into a message and "function-ships" that request to a Control and I/O Daemon (CIOD) running on an I/O node. For example, a write system call sends a message containing the file descriptor number, length of the buffer, and the buffer data. As illustrated in Figure 2, CIOD retrieves messages from the collective network and directs them to an ioproxy program using a shared buffer. Each ioproxy process is associated with a specific process on a compute node. The ioproxy's filesystem state mirrors the CNK process's state (e.g., file seek offsets, current working directory, user/group permissions). The ioproxy decodes the message, demarshals the arguments, and performs the system call that was requested by the compute node process. When

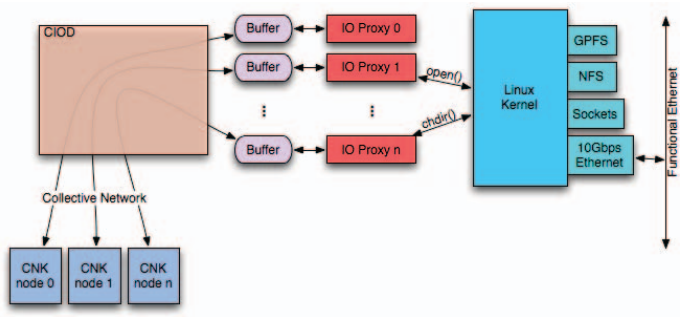

Fig. 2. The CNK ⇔ CIOD Function-Shipped Protocol

the system call completes, the results are marshaled and sent back to the compute node that originated the request.

From the Linux perspective, the ioproxies perform standard I/O operations, e.g., a filesystem operation from CNK behaves as if it was performed from Linux (although the blocking of data is different due to the collective network and CIOD protocol). The calls produce the same result codes, network filesystem nuances, etc. Additionally, filesystems that are installed on the I/O nodes (such as NFS, GPFS, PVFS, Lustre) are available to CNK processes via the ioproxy. Thus, in a lightweight and low-noise manner, CNK can provide the full suite of I/O system calls available to Linux.

Our experiences with the I/O offload strategy, and in particular the 1-to-1 mapping of ioproxies to Compute Node (CN) processes have been positive. The amount of code required in CNK to implement the offload is minimal, and running Linux allows us to easily inherit all of the POSIX semantics.

B. CNK Runtime Support

Unlike on BG/L, on BG/P we had a design objective to use glibc unmodified. The goal was to unify the toolchains between a standard Linux software stack and the CNK software stack, resulting in less maintenance and better testing coverage. In this section we describe how we leveraged open-source implementations to provide the core CNK support for Linux-compatible threading and dynamic linking.

1) Threading: We examined what it would take to use the NPTL threading package in glibc on CNK. An investigation showed it required only a small number of system calls beyond our current set. A subset of both clone and set_tid_address were needed for thread creation (e.g., pthread_create). For atomic operations, such as pthread_mutex, a full implementation of futex was needed. For thread signaling and cancellation, we needed to implement sigaction. Although we probably could have had a basic custom threading package implemented sooner, by leveraging NPTL, CNK provides a full-featured pthread package that is well understood by application developers.

This path was not without concerns. One was that Linux uses clone support for both thread and process creation. We analyzed the glibc code and determined that glibc uses the clone system call with a static set of flags. The flags to clone are validated against the expected flags, but we did not need to reference the flags elsewhere in the kernel. Other

parameters to clone included the child's stack and thread local storage pointers, as well as the child-parent thread IDs. The glibc library performs a uname system call to determine the kernel capabilities so we set CNK's version field in uname to 2.6.19.2 to indicate to glibc that we have the proper support. For stack creation, glibc uses standard malloc calls to allocate the storage. Many stack allocations exceed 1MB, invoking the mmap system call as opposed to brk. However, CNK supports both brk and mmap, so this is not an issue.

One of the simplifications we made to the threading support was in the thread scheduler. Unlike Linux and other full-featured kernels, CNK provides a simple non-preemptive scheduler, with a small fixed number of threads per core.

2) Dynamic Linking: On BG/P we support Python. Although Python is an interpreted language, it can be extended via libraries. Traditionally, those libraries are demand-loaded through dynamic libraries. One option to provide dynamic linking was to merge the application's dynamic libraries outside of the compute nodes as an additional step in job launch. This would have been simple, but may not have been practical because dlopen()-type functionality would be needed. Another option was to support the ld.so dynamic linker from glibc or implement a dynamic linker similar to it. Similar to our analysis of NPTL, we determined that ld.so did not need many system calls in order to achieve functionality, and again by going this path we provided a more standard and well-understood solution. Concretely, ld.so needed to statically load at a fixed virtual address that was not equal to the initial virtual addresses of the application, and ld.so needed MAP_COPY support from the mmap() system call.

One of the simplifications we made was that a mapped file would always load the full library into memory, rather than page-faulting many small pages across the network. We also decided not to honor page permission settings, i.e., read, write, or execute, on the dynamic library's text/read-only data. For example, applications could therefore unintentionally modify their text or read-only data. This was a conscious design decision consistent with the lightweight philosophy. Providing this permission support would have required dynamic page misses and faulting pages from across networked storage. This would have significantly increased complexity of the kernel and introduced noise. By loading the entire library into memory at load-time, this OS noise is contained in application startup or use of dlopen and can be coordinated between nodes by the application.

C. Memory Management

Most operating systems maintain logical page tables and allow for translation misses to fill in the hardware page tables as necessary. This general solution allows for page faults, a fine granularity of permission control, and sharing of data. There are, however, costs to this approach. For example, there is a performance penalty associated with the translation miss. Further, translation misses do not necessarily occur at the same time on all nodes, and become another contributor of OS noise. Another complication arises from the power-efficient network hardware that does not implement sophisticated page translation facilities.

To meet performance and simplicity goals, CNK implements a memory translation mapping that is static for the duration of the process. A process can query the static map during initialization and reference it during runtime without having to coordinate with CNK. In order to keep the mapping simple, CNK implements the following four address ranges that are contiguous in physical memory: 1) text (and read-only data) (.text, .rodata), 2) data (globals) (.bss, .data), 3) heap and stack, and 4) shared memory.

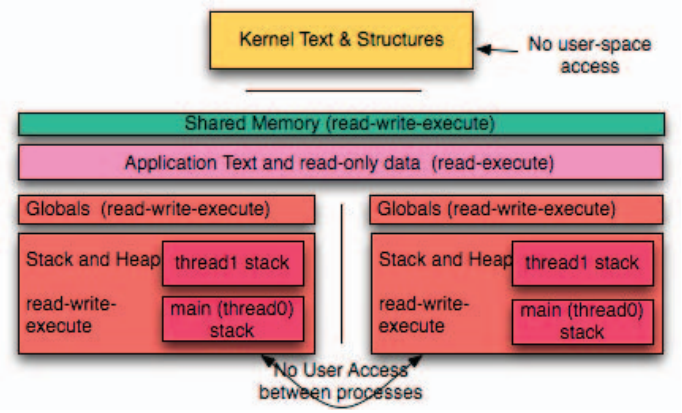

Fig. 3. CNK Memory Layout

When an application is loaded, the ELF (Executable and Linkable Format) section information of the application indicates the location and size of the text and data segments. The number of processes per node and size of the shared memory region are specified by the user. This information is passed into a partitioning algorithm, which tiles the virtual and physical memory and generates a static mapping that makes effective use of the different hardware page size (1MB, 16MB, 256MB, 1GB) and that respects hardware alignment constraints.

During application execution, memory may be allocated via the standard brk and mmap system calls. The mmap system call tracks which memory ranges have been allocated. It also coalesces memory when buffers are freed, or permissions on those buffers change. However, since CNK statically maps memory, the mmap system call does not need to perform any adjustments, or handle page faults. It merely provides free addresses to the application. With this strategy, glibc could be enhanced to perform all of the memory management in user space, but that would have led to a customized version of glibc.

A useful memory protection feature is a guard page to prevent stack storage from descending into heap storage, see Figure 4. CNK provides this functionality by using the Blue Gene Debug Address Compare (DAC) registers. The guard range is determined by the application. The glibc NPTL library performs a mprotect() system call prior to the clone(). CNK remembers the last mprotect range and makes an assumption during the clone syscall that the last mprotect applies to the new thread. The guard page covering the main thread is special. It resides on the heap boundary and a memory allocation performed by another thread could move the heap boundary. That newly allocated storage could be legitimately

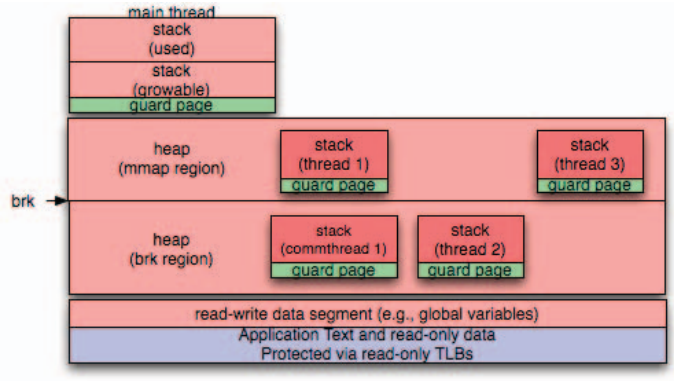

Fig. 4. Guard Page Functionality in CNK

referenced by the main thread. So when the heap boundary is extended, CNK issues an inter-processor interrupt to the main thread in order to reposition the guard area.

D. Persistent memory

Data management is becoming increasingly important to HPC applications as the relative cost of storage bandwidth to flops increases. A standard HPC data model is for an application to start with a clean memory slate, load the data from disk, perform computation, and store the data to disk. For consecutive applications accessing the same data, this can add unnecessary overhead. To address this challenge, on BG/P, we developed a feature that allows an application to tag memory as persistent. When the next job is started, memory tagged as persistent is preserved, assuming the correct privileges. The application specifies the persistent memory by name, in a manner similar to the standard shm_open()/mmap() methods. One important feature for persistent memory is that the virtual addresses used by the first application are preserved during the run of the second application. Thus, the persistent memory region can contain linked-list-style pointer structures. The persistent memory could also be used by an application in a manner similar to ramfs offered by Linux.

V. Low Noise, Functionality, and Performance Stability of CNK

Although this paper focuses on CNK's design and experiences with it, in this section we provide the performance results of that design. In particular, we present results demonstrating that CNK yields very low noise (contribution 2a), describe a set of applications that run on CNK unmodified demonstrating (contribution 3), describe the high performance achieved through CNK's design by the messaging layers, and we finish by showing the performance stability of CNK on sample applications (contribution 2a). Throughout the section we describe the features of CNK that would be more challenging on an FWK (contribution 2a).

A. Low Noise

OS jitter or noise in a large parallel system is magnified when an application synchronizes across a large number of nodes. Delays incurred by the application at random times each cause a delay in an operation, and at large scale many nodes compound the delay causing a noticeable performance impact[24]. There has been a lot of work on understanding noise and more recent characterizations[13] have described the salient characteristics of noise that affect applications performance. Potential techniques to address this problem are 1) coordinate the delays so they are not compounded by scale, or 2) eliminate the delays. There are limits to how effective the former is, and its effectiveness is also application dependent. CNK takes the latter approach.

One way to measure the noise of a kernel is performed by the FWQ (Fixed Work Quanta) benchmark[8]. This is a single node benchmark, i.e., no network communication, that measures a fixed loop of work that, without noise, should take the same time to execute for each iteration.

The configuration we used for CNK included 12,000 timed samples of a DAXPY (double precision ax + y linear algebra routine) on a 256 element vector that fits in L1 cache. The DAXPY operation was repeated 256 times to provide work that consumes approximately 0.0008 seconds (658K cycles) for each sample on a BG/P core. This is performed in parallel by a thread on each of the four cores of a Blue Gene/P node.

Figures 5 through 7 show the results from running the FWQ on a BG/P node. The graphs show the number of cycles (Y axis) it takes to complete a given iteration (X axis). Ideally, all iterations would take the same number of cycles at the minimum value of 658,958. The figures plots the cycle times for each the 12,000 samples. The left graph, Figure 5, shows the values for Linux, the middle graph, Figure 6, plots the data for CNK with the same Y axis, and right graph, Figure 7, is the same data with a considerably zoomed in Y axis to show detail of the CNK values. The node was running Linux based on SUSE kernel version 2.6.16. Efforts were made to reduce noise on Linux; all processes were suspended except for init, a single shell, the FWQ benchmark, and various kernel daemons that cannot be suspended.

The minimum time on any core for any iteration was 658,958 processor cycles. This value was achieved both on Linux and on CNK. The CNK graphs demonstrate the low noise achievable with an LWK strategy. The maximum variation is less than 0.006%. For Linux the maximum cycle time varied by 38,076 cycles on core 0, 10,194 cycles on core 1, 42,000 cycles on core 2 and 36,470 cycles on core 3. This is variation is greater than 5% on cores 0, 2 and 3.

The impact of noise reduction on application performance is not necessarily a linear mapping. Small amount of noise may not affect behavior, while moderate amounts may have an additive impact. Other work[13] has done a good job characterizing the impact of noise on application performance.

B. Functionality

One indication of CNK functionality is the applications it supports. OpenMP-based benchmarks such as AMG, IRS, and SPhot run threaded on CNK without modification. The UMT benchmark also runs without modification, and it is driven by a Python script, which uses dynamic linking. UMT also uses

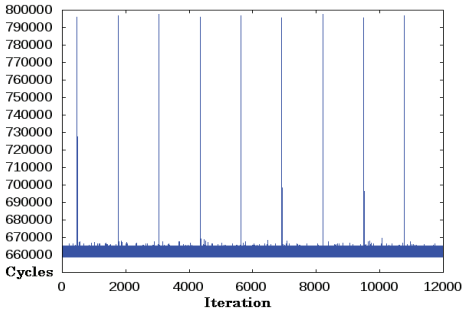

Fig. 5. FWQ for Linux core 0

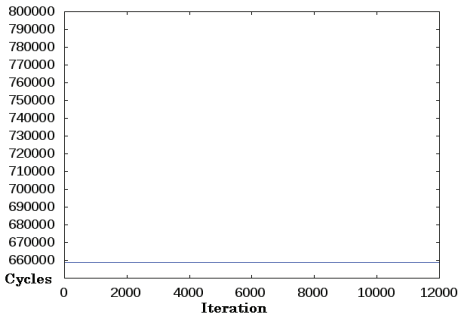

Fig. 6. FWQ for CNK core 0

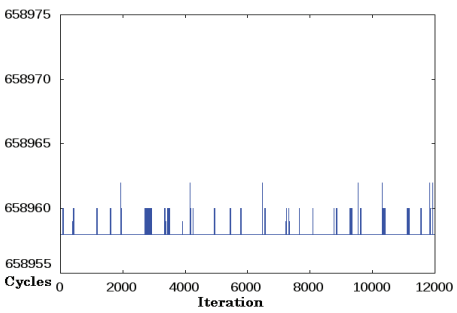

Fig. 7. FWQ for CNK core 0 zoomed Y axis

Protocol	Latency(μs)
DCMF Eager One-way	1.6
MPI Eager One-way	2.4
MPI Rendezvous One-way	5.6
DCMF Put	0.9
DCMF Get	1.6
ARMCI blocking Put	2.0
ARMCI blocking Get	3.3

TABLE I
LATENCY FOR VARIOUS PROGRAMMING MODELS IN SMP MODE

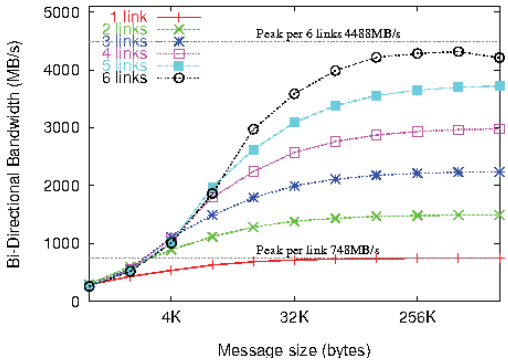

Fig. 8. Throughput of rendezvous protocol for near-neighbor exchange

OpenMP threads. FLASH, MILC, CPS, Chroma, NEK, GTC, DOCK5/6, QBOX, MGDC, RXFF, GMD, DNS3D, HYPO4D, PLB, LAMMPS, and CACTUS are known to scale on CNK to more than 130,000 cores.

Additional functionality for unique hardware features was demonstrated in an earlier version of CNK to support the 2007 Gordon Bell Prize for "Kelvin-Helmholtz instability in molten metals."[15] CNK was able to handle L1 parity errors by signaling the application with the error to allow the application to perform recovery without need for heavy I/O-bound checkpoint/restart cycles.

C. Achieving High Performance for System Software

Another important metric is how well and with how much effort other system software can achieve high performance. A key performance area for HPC software is messaging performance. Some applications are latency sensitive due to a reliance on many short messages, while others' performance depends on achieving high bandwidth. The Blue Gene DCMF (Deep Computing Messaging Framework) relies on CNK's ability to allow the messaging hardware to be used from user space, the ability to know the virtual to physical mapping from user space, and the ability to have large physically contiguous chunks of memory available in user space. Data taken from previous work[20] is shown in Table I that illustrates low latency obtained through the use of user space accessibility, and Figure 8 that shows DCMF achieving maximum bandwidth by utilizing large physically contiguous memory. These came effectively for free with CNK's design and implementation, but modifying a vanilla Linux, especially to provide large physically contiguous memory, would be difficult.

D. Performance Stability

To demonstrate performance stability we ran 36 runs of LINPACK on Blue Gene/P racks. Each rack produced 11.94 TFLOPS. The execution time varied from 16080.89 seconds to 16083.00 seconds, for a maximum variation of 2.11 seconds (.01%) over a 4 hour and 28 minute run and a standard deviation of less than 1.14 seconds. Another example of repeatability is demonstrated by repeated execution of a benchmark under CNK compared to the same benchmark under Linux. This experiment measured the performance of the mpiBench_Allreduce test, which is a test in the Phloem benchmark suite[9]. The test measured the time to perform a double-sum allreduce on 16 Blue Gene/P nodes over one million iterations. Over this time the test produced a standard deviation of 0.0007 microseconds (effectively 0, likely a floating point precision error).

A similar test was performed with Linux on Blue Gene/P I/O nodes interconnected by 10Gbps ethernet. Background daemons where suspended as allowed, but NFS was required to capture results between tests. This test was the same double-sum allreduce, but executing on only 4 Blue Gene/P I/O nodes over 100,000 iterations. The Linux test was executed twenty times and produced a standard deviation of 8.9 microseconds.

Whether or not this level of variance has a significant application impact would need more study, but application groups[11] have found that BG/P's performance stability allowed them to more easily and more effectively tune their applications

VI. WHAT IS NOT IN CNK

CNK is designed to be simple. We have consciously limited the functionality of CNK to stay consistent with our scalability design philosophy. This philosophy makes kernel implementation feasible for a small team and improves its reliability. Simplification is performed across the major components of

the kernel. However, there are applications for which the functionality is not sufficient and need a more complete set of Linux functionality. In this section we describe what is not in CNK and in the next section describe the pros and cons of the boundaries we have drawn and describe how easy or difficult it is to achieve a given functionality in Linux and CNK.

A. I/O Subsystem

The I/O subsystems in CNK are virtually non-existent. This is accomplished by function shipping the requests to a dedicated I/O node. Correct semantics are difficult to achieve in POSIX I/O and would take considerable time to stabilize if they were implemented in CNK. This is true for both network filesystems and standard networking protocols such as TCP/IP. Delegating the filesystem to a separate Linux node means that complex problems such as filesystem caching, readahead, writebehind, client consistency, and scalability need not be implemented in CNK. CNK also does not need to provide a virtual filesystem layer or module loading layer to support such filesystems, nor do filesystems need to be ported to CNK. Linux provides this functionality and leverages a large user and development community for finding and fixing problems. Also, a driver layer for the Blue Gene specialized network hardware is unnecessary because this hardware is fixed on Blue Gene. Although this does keep CNK simple, there are some consequences. For example, to mmap a file, CNK copies in the data and only allows read-only access.

B. Memory Management

Much of CNK's memory management architecture is driven by the goal of providing static TLB mapping for an application's address space. CNK does not implement demand paging, or copy-on-write, and as a filesystem is not implemented by CNK there is no need to implement a unified page cache between the virtual memory manager and filesystem. Further, CNK has simplified its own use of memory by placing strict limits on sizes of kernel data structures, allocating all of its structures statically. CNK does not maintain a heap of intermingled data structures throughout physical memory. This simple strategy for memory management makes debugging memory corruption problems easier, and makes protection of DMA directly programmed by application code straightforward.

C. Scheduling

Thread scheduling under CNK is non-preemptive with fixed affinity to a core. The "scheduler" has a simple decision limited to threads sharing a core when a thread specifically blocks on a futex or explicitly yields. Sharing a core is rare in HPC applications so generally a thread enters the kernel only to wait until a futex may be granted by another core rather than yield to a thread on the same core. I/O function shipping is made trivial by not yielding the core to another thread during an I/O system call. This has the side effect of never switching kernel context during execution of a system call on a kernel stack. Instead, the scheduler only has to consider the case of context switching user state.

VII. Pros and Cons for HPC Applications

The design point we chose for CNK has its advantages and disadvantages. In this section we describe the pros and cons we have had from our experiences with CNK and then how easy or difficult it is to achieve a given functionality in Linux and CNK.

A. Pros for HPC Applications

Many of the design simplifications in CNK enhance the performance of HPC applications without requiring application effort. CNK provides strict processor affinity for processes and threads. This avoids unnecessary context switch overhead and improves L1 cache use. As this is common in HPC application, this "limitation" rarely has a negative impact, and instead relieves the application of affinity responsibility. Similarly, CNK pins memory with huge TLB entries to avoid TLB misses. Using huge TLB pages in Linux is often a non-trivial enhancement to an application, especially if the huge pages are used to back the application code or stack or are requested on-the-fly, for example for messaging. CNK requires no application modification to take advantage of the large pages. Linux has become easier over time, but still requires tuning and is not automatic. Another advantage of the memory layout provided by CNK is that nearly the full 4GB 32-bit address space of a task can be mapped on a 32-bit processor. Linux typically limits a task to 3GB of the address space due to 32-bit limitations. While this was on issue on BG/P, on next generation Blue Gene hardware, with 64-bit processors, it will no longer be.

Simple memory mappings allow CNK applications to directly drive the DMA torus hardware without concern of corrupting the kernel. This results in simplified hardware and improved DMA latency because a special I/O memory management unit and the related system call overhead for setting up communication windows is unnecessary. Function-shipping the I/O system calls provides up to two orders of magnitude reduction in filesystem clients. Finally, the simplicity of CNK allows it to initialize quickly and makes it easier to provide cycle reproducible debugging.

B. Cons for HPC Applications

There are downsides to the simplification of CNK. The strict affinity enforced by the scheduler does not allow applications to use threads in creative ways. For example, it is not possible to run low-priority background threads while a primary thread performs compute work. CNK also does not allow a node to be divided non-uniformly. MPI cannot spawn dynamic tasks because CNK does not allow fork/exec operations. Some applications overcommit threads to cores for load balancing purposes, and the CNK threading model does not allow that, though Charm++ accomplishes this with a user-mode threading library.

In order to provide static mapping with a limited number of TLB entries, the memory subsystem may waste physical memory as large pages are tiled together. The dynamic linker does not protect read-only and text sections of dynamic

Description	CNK	Linux
Large page use	easy	medium
Using multiple large page sizes	easy	medium[1]
Large physically contiguous memory	easy	easy - hard[2]
No TLB misses	easy	not avail
Full memory protection	not avail	easy
General dynamic linking	not avail	easy
Full mmap support	not avail	easy
Predictable scheduling	easy	medium
Over commit of threads	easy - not avail[3]	medium
Performance reproducible	easy	medium - hard
Cycle reproducible execution	easy	not avail

TABLE II
EASE OF USING DIFFERENT CAPABILITIES IN CNK AND LINUX

Description	CNK	Linux
Large physically contiguous memory	avail	medium
No TLB misses	avail	hard
Full memory protection	medium	avail
General dynamic linking	medium	avail
Full mmap support	hard	avail
Cycle reproducible execution	avail	medium

TABLE III
EASE OF IMPLEMENTING CAPABILITIES IN CNK AND LINUX

libraries loaded after the application starts. The lack of a unified page cache means that pages of code and read-only data can not be discarded when memory pressure is high. The lack of a page cache also means that dynamic objects are not shared between tasks that physically share a node.

Other disadvantages include that CNK divides memory on a node evenly among the tasks on the node. If one task's memory grows more than another, the application could run out of memory before all the memory of a node was consumed. Also, CNK requires the user to define the size of the shared memory allocation up-front as the application is launched. Finally, the application cannot take advantage of the scripting environment offered by a full featured operating system; an application cannot be structured as a shell script that forks off related executables.

C. Ease of functionality for CNK versus Linux

In this section we combine and summarize the previous sections on design and experience. Table II lists a series of mechanisms, capabilities, and requirements that HPC applications may be interested in using. Columns two and three indicate how difficult it is to use that feature in each of the systems: easy, medium, or hard. For the features that are listed as not-avail in Table II, Table III indicates the difficulty of implementing them in that OS. The Linux that was evaluated was from the 2.6.30 generation, and CNK is BG/P's CNK.

[1] multiple page sizes just became available in Linux

[2] it is easy to request, but depending on memory layout may not be granted

[3] BG/P introduced three threads per core in November, next generation CNK is planned to have a variable number available at compile time

VIII. MOVING FORWARD WITH CNK

We have been working on the software design for the next generation machine over the last couple years. For the kernel, we plan to continue with the LWK approach with CNK. Given the increasing demands on the system software ecosystem, this decision was made only after considerable examination. Given the trends in multicore and multithreaded architectures, as indicated above, we introduced the capability to have multiple software threads per core on BG/P. To handle this we are moving to an extended thread affinity model. This model provides a good example of the tensions, which we are always balancing, between the LWK philosophy and the desire to support general computing models.

A. Extended Thread Affinity Control

Until recently on BG/P, CNK allowed only one software thread (pthread) per core. In preparation for the next generation machine, this was relaxed to multiple pthreads per core. However, the assignment of cores to processes is set statically at job launch time, and a given core executes only on behalf of the process to which it is assigned. This static assignment clashes with a programming model in which programs alternate between phases with many processes executing in parallel and phases with a few processes using many threads. A specific example is an application that starts with n MPI tasks per node, one per core, and then enters an OpenMP phase in which one of the processes wants to use all the cores. Under Linux this can easily be accommodated through standard threading mechanisms.

The exploration of possible thread-affinity extensions in CNK is representative of other design explorations and illustrates how maintaining an LWK philosophy while providing familiar functionality can be challenging. Supporting a fully general thread-affinity model would prevent the static TLB mapping that is a hallmark of the LWK design, and would lead to questions of ownership for such hardware subsystems as the messaging unit. Instead we plan an extension that allows a given core to alternate between executing a pthread from its assigned process and executing a pthread from a single designated "remote" process. By partnering with and understanding the needs of our application developers, we were able to put together a thread-affinity extension that supports the actual usage models that programmers need while staying within the design philosophy of CNK.

IX. CONCLUSIONS

We described the salient new features of BG/P's CNK and our experiences with them. We showed that we could keep the lightweight kernel approach, but still provide a Linux-like operating environment. We did this by leveraging open source Linux components including glibc and the NPTL threading package, and function shipping I/O requests from the compute nodes to the I/O nodes. We demonstrated the resultant kernel has extremely low noise. It is reliable and scales well to hundreds of thousands of nodes.

The interesting exploration of lightweight, Linux-based, or other approaches will continue, with advances in one approach

positively benefiting the others. Our solution marks another interesting point in the space of possible solutions. Because CNK's code base is open source[1], it provides the opportunity for the community to leverage and participate in the on-going effort. Whether the exascale answer will be an LWK or FWK or more likely a mixture remains to be seen. The description in this paper serves to bring forward tradeoffs that will be important as the system's community investigates the needs of exascale computing.

ACKNOWLEDGMENTS

A kernel development project is a large undertaking, and requires the efforts of many people. We would like to thank many of the other developers who have contributed to the kernel including John Attinella, Michael Mundy, Thomas Musta, Bryan Rosenburg, Richard Shok, and Andy Tauferner. We would like to thank Roy Musselman and the performance team for their help with gathering the FWQ and application data. Thank you to Sameer Kumar and the messaging team for the DCMF performance data. We would also like to thank Philip Heidelberger and Martin Ohmacht for their help and insight with many of the low-level issues and performance. Thank you to Bryan Rosenburg for his time in editing; his many suggestions have improved the paper's quality and readability. Thank you to Craig Stunkel for his careful read of this document.

REFERENCES

[1] Argonne's Blue Gene/P open source repository. http://wiki.bg.anl-external.org/index.php.
[2] International exascale software project. www.exascale.org.
[3] Kitten lightweight kernel. https://software.sandia.gov/trac/kitten.
[4] The l4 microkernel. http://ertos.nicta.com.au/research/l4/.
[5] NASA's Pleiades Machine. http://www.nas.nasa.gov/Resources/ Systems/pleiades.html.
[6] Red Storm web page. http://www.sandia.gov/ASC/redstorm.html.
[7] SGI's Altix Family. http://www.sgi.com/products/servers/altix/.
[8] The FTQ/FWQ Benchmark at LLNL. https://asc.llnl.gov/sequoia/benchmarks/ FTQ_summary_v1.1.pdf.
[9] The Phloem Benchmark at LLNL. https://asc.llnl.gov/sequoia/ benchmarks/PhloemMPIBenchmarks_summary_v1.0.pdf.
[10] ZeptoOS: The small linux for big computers. http://www.mcs.anl.gov/research/projects/zeptoos/.
[11] S. Alam, R. Barrett, M. Bast, M. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. Vetter, P. Worley, and W. Yu. Early evaluation of BlueGene/P. In *Supercomputing*, 2008.
[12] R. Brightwell. Catamount N-Way Lightweight Kernel. http://www.sandia.gov/mission/ste/r&d100/2009winners/CNWFinal.pdf.
[13] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *Supercomputing*, 2008.
[14] M. E. Giampapa, R. Bellofatto, M. A. Blumrich, D. Chen, M. B. Dombrowa, A. Gara, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, B. J. Nathanson, B. D. Steinmacher-Burow, M. Ohmacht, V. Salapura, and P. Vranas. Blue gene/l advanced diagnostics environment. *IBM Journal for Research and Development*, 49(2), 2005.
[15] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz. Extending stability beyond cpu millennium a micron-scale atomistic simulation of kelvin-helmholtz instability. In *Supercomputing*, Reno, November 2007.
[16] E. V. Hensbergen, C. Forsyth, J. McKie, and R. Minnich. Petascale Plan 9 on Blue Gene. In *Usenix 2007*, Santa Clara, CA, June 17-22 2007.
[17] IBM Blue Gene Team. Overview of the Blue Gene/P project. *IBM Journal for Research and Development*, 52(1/2), January 2008.
[18] L. S. Kaplan. Lightweight Linux for high-performance computing. Linux World.com, http://www.linuxworld.com/news/2006/120406-lightweight-linux.html, December 2006.
[19] S. Kelly and R. Brightwell. Software architecture of the lightweight kernel, Catamount. In *Cray Users' Group Annual Technical Conference*, Albuquerque, New Mexico, June 2005.
[20] S. Kumar, G. Dozsa, G. Almasi, D. Chen, M. E. Giampapa, P. Heidelberger, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. Archer. The deep computing messaging framework: Generalized scalable message passing on the Blue Gene/P Supercomputer. In *ICS*, June 2008.
[21] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios: A new open source virtual machine monitor for scalable high performance computing. In *IPDPS*, April 2010.
[22] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, S. Jaconette, M. Levenhagen, R. Brightwell, and P. Widener. Palacios and Kitten: High performance operating systems for scalable virtualized and native supercomputing. Technical report, EECS Northwestern University, July 2009.
[23] J. Moreira, M. Brutman, J. Castanos, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt. Designing a highly-scalable operating system. In *Supercomputing*, November 2006.
[24] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing 2003*, November 2003.
[25] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, April 2009.
[26] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *22nd ACM International Conference on Supercomputing*, Island of Kos, Aegean Sea, Greece, June 7-12 2008.
[27] D. Wallace. Compute Node Linux: Overview, progress to date & roadmap. http://www.nccs.gov/wp-content/uploads/2007/08/wallace_paper.pdf, 2007.