A lightweight virtual machine monitor for Blue Gene/P

Jan Stoess, Udo Steinberg, Volkmar Uhlig, Jens Kehne, Jonathan Appavoo and Amos Waterland

International Journal of High Performance Computing Applications 2012 26: 95 originally published online 27 March 2012
DOI: 10.1177/1094342011434815

THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

Article

The International Journal of High Performance Computing Applications
26(2) 95–109

DOI: 10.1177/1094342011434815
hpc.sagepub.com

A lightweight virtual machine monitor for Blue Gene/P

Jan Stoess[1,3], Udo Steinberg[2], Volkmar Uhlig[3], Jens Kehne[1], Jonathan Appavoo[4] and Amos Waterland[5]

Abstract

In this paper, we present a lightweight, micro-kernel-based virtual machine monitor (VMM) for the Blue Gene/P supercomputer. Our VMM comprises a small μ-kernel with virtualization capabilities and, atop, a user-level VMM component that manages virtual Blue Gene/P cores, memory, and interconnects; we also support running native applications directly atop the μ-kernel. Our design goal is to enable compatibility to standard operating systems such as Linux on BG/P via virtualization, but to also keep the amount of kernel functionality small enough to facilitate shortening the path to applications and lowering operating system noise.

Our prototype implementation successfully virtualizes a Blue Gene/P version of Linux with support for Ethernet-based communication mapped onto Blue Gene/P's collective and torus network devices. Our first experiences and experiments show that our VMM still shows a substantial performance hit, and that support for native application environments is a key requirement towards fully exploiting the capabilities of a supercomputer. Altogether, our approach poses an interesting operating system alternative for supercomputers, providing the convenience of a fully featured commodity software stack, while also promising to deliver the scalability and low latency of an HPC operating system.

Keywords

Operating systems, virtualization, micro-kernels, lightweight kernels, supercomputing, high-performance computing

1 Introduction

A substantial fraction of supercomputer programmers today write software using a parallel programming runtime such as MPI on top of a customized lightweight kernel. For Blue Gene/P (BG/P) machines in production, IBM provides such a lightweight kernel called Compute Node Kernel (CNK) (Giampapa et al., 2010). CNK runs tasks massively parallel, in a single-thread-per-core fashion. Like other lightweight kernels, CNK supports a subset of a standardized application interface (POSIX), facilitating the development of dedicated (POSIX-like) applications for a supercomputer. However, CNK is not fully POSIX-compatible: it lacks, for instance, comprehensive scheduling or memory management as well as standards-compatible networking or support for standard debugging tools. CNK also supports I/O only via function-shipping to I/O nodes.

CNK's lightweight kernel model is a good choice for the current set of BG/P HPC applications, providing low operating system (OS) noise and focusing on performance, scalability, and extensibility. However, today's HPC application space is beginning to scale out towards exascale systems of *truly* global dimensions, spanning companies, institutions, and even countries. The restricted support for standardized application interfaces of lightweight kernels in general, and CNK in particular, renders porting the sprawling diversity of scalable applications to supercomputers more and more a bottleneck in the development path of HPC applications.

In this paper, we explore an alternative, hybrid OS design for BG/P: a μ-kernel-based virtual machine monitor (VMM). At the lowest layer, in kernel mode, we run a μ-kernel that provides a small set of basic OS primitives for constructing customized HPC applications and services at user level. We then construct a user-level VMM that fully

This research was partly conducted by the authors while at IBM Watson Research Center, Yorktown Heights, NY.
[1]Faculty of Informatics, Karlsruhe Institute of Technology, Germany
[2]Department of Computer Science, Technische Universität Dresden, Germany
[3]HStreaming LLC, USA
[4]Department of Computer Science, Boston University, USA
[5]Harvard School of Engineering and Applied Sciences, USA

Corresponding author:
Jan Stoess, Faculty of Informatics, Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
Email: jan.stoess@kit.edu

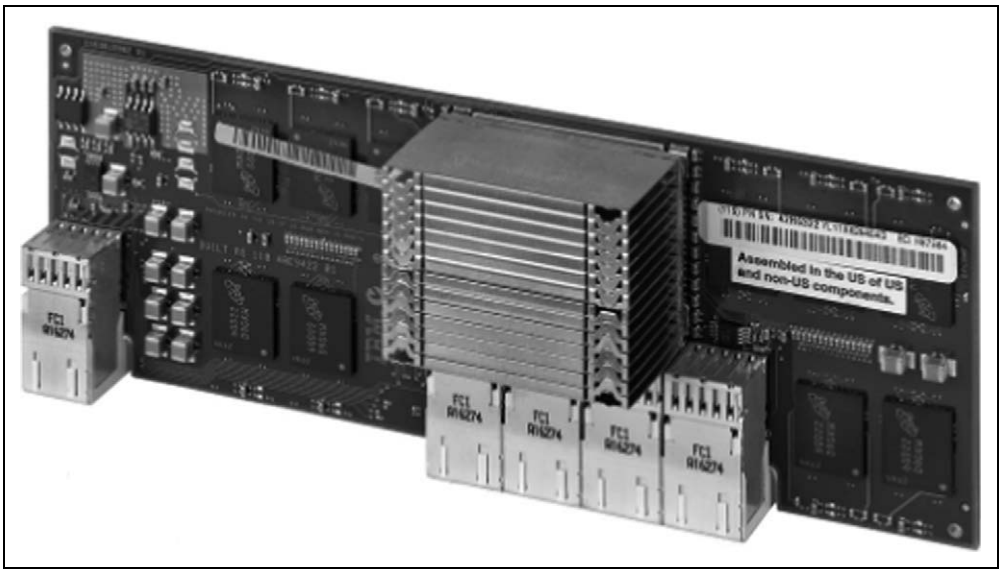

Figure 1. Blue Gene/P compute node card.

virtualizes the BG/P platform and allows arbitrary Blue Gene OSs to run in virtualized compartments. We finally construct a native communication library that directly interfaces with BG/P's high-performance torus interconnect, demonstrating the benefits of a μ-kernel to native HPC application development.

The benefits of a μ-kernel-based VMM architecture are twofold: on the one hand, it provides compatibility with BG/P hardware, allowing programmers to ship the OS they require for their particular applications along with it, like a library. For instance, our VMM successfully virtualizes a Blue Gene version of Linux with support for Ethernet-based communication, allowing virtually any general-purpose Linux application or service to run on BG/P. On the other hand, our μ-kernel also resembles the light-weight kernel approach in that it reduces the amount of kernel functionality to basic resource management and communication. Those mechanisms are available to native applications running directly on top of the μ-kernel, and programmers can use them to customize their HPC applications for better efficiency and scalability, and to directly exploit the features of the tightly interconnected BG/P hardware. However, μ-kernel and VMM architectures also imply a potential penalty for efficiency, as they increase kernel-user interaction and add another layer of indirection to the system software stack. Nevertheless, the need for standardized application interfaces is becoming more prevalent, and we expect our work to be an insightful step towards supporting such standardization on supercomputer platforms.

Altogether, our architecture strives to facilitate a path for easy development and porting of applications to the supercomputer world, where programmers can run a virtualized version of any general-purpose, scalable, and distributed application on Blue Gene without much hassle; but where they can also customize those applications for better efficiency and scalability, with help from the μ-kernel's native interface.

The idea of a virtualization layer for HPC systems is not new (Lange et al., 2010), nor is the idea of using a decomposed VMM architecture to deliver predictable application performance (Heiser and Leslie, 2010). However, to our knowledge, this is the first approach to provide a commodity system software stack and hide the hardware peculiarities of a highly customized HPC architecture such as BG/P, while still being able to run hand-optimized code side-by-side. The initial results are promising: Our prototype based on the L4 μ-kernel fully virtualizes BG/P compute nodes and their high-performance network interconnects, and successfully runs multiple instances of a BG/P version of Linux.

The rest of the paper is structured as follows: Section 2 presents the basic architecture of our μ-kernel-based virtualization approach for BG/P. Section 3 presents details of the μ-kernel and Section 4 presents details of our user-level VMM component. Section 5 gives details of native application support. Finally, Section 6 presents an initial evaluation of our prototype, followed by related work in Section 7 and a summary in Section 8.

2 System overview

In this section, we present the design of our decomposed VMM for BG/P. We start with a very brief overview of the BG/P supercomputer: The basic building block of BG/P is a compute node, which is composed of an embedded quad-core PowerPC, five networks, a DDR2 controller, and either 2 or 4 GB of RAM, integrated into a system-on-a-chip (Figure 1). One of the largest BG/P configurations is a 256-rack system of two mid-planes each, which, in turn, comprise 16 node cards with 32 nodes each, totaling over 1 million cores and 1 PB of RAM. BG/P features three key

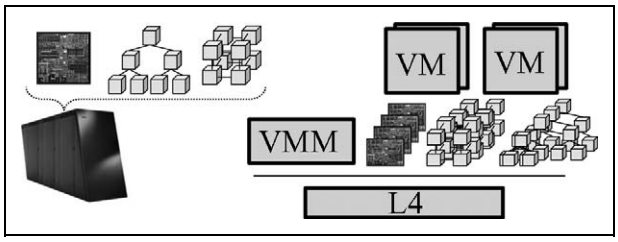

Figure 2. A μ-kernel based VMM virtualizing BG/P cores and interconnects.

communication networks, a torus, and a collective interconnect, and an external 10GE Ethernet network on I/O nodes.

The basic architecture that we used is illustrated in Figure 2. We use an enhanced version of the L4 μ-kernel as the privileged supercomputer kernel (Liedtke, 1995). Our BG/P implementation uses a recent L4 version named L4Ka::Pistachio (L4 Development Team, 2009). Traditional hypervisors such as Xen (Pratt et al., 2005) or VMware (Agesen et al., 2010) are virtualization-only approaches in the sense that they provide virtual hardware – virtual CPUs, memory, disks, networks, etc. – as *first class* abstractions. L4 is different in that it offers a limited set of OS abstractions to enforce safe and secure execution: threads, address spaces, and inter-process communication (IPC). These abstractions are at a sufficiently low level to allow the construction of an efficient virtualization layer atop, as has been demonstrated on commodity systems (Härtig et al., 1997; Le Vasseur et al., 2004).

While L4 provides core primitives, the actual VMM functionality is implemented as a user-level application outside the privileged kernel. The basic mechanics of such an L4-based VMM is as follows: L4 merely acts as a safe messaging system propagating sensitive guest instructions to a user-level VMM. That VMM, in turn, decodes each instruction, emulates it appropriately and then responds with a fault reply message that instructs L4 to update the guest VM's context and then to resume guest VM execution.

Our decomposed, μ-kernel-based VMM architecture has benefits that are particularly interesting on HPC systems: Since L4 provides a minimal yet sufficiently generic abstraction, it also supports native applications that can bypass the complete virtualization stack whenever they need performance. Hybrid configurations are also possible: An application can be started within a guest VM, with access to all legacy services of the guest kernel; for improved efficiency, it can later choose to employ a native HPC library (e.g. an MPI library running directly atop L4). In the following, we will first describe how L4 facilitates running a VMM atop; we will then describe the user-level VMM part in the subsequent section.

3 A micro-kernel with virtualization capabilities

In our architecture, L4 acts as the privileged part of the VMM, responsible for partitioning processors, memory, device memory, and interrupts. Our virtualization model is largely common to L4's normal execution model. We use: (a) L4 threads to virtualize the processor; (b) L4 memory-mapping mechanisms to provide and manage guest-physical memory, and (c) L4 IPC to allow emulation of sensitive instructions through the user-level VMM. However, in virtualization mode, threads and address spaces have access to an extended instruction set architecture (ISA) and memory model (including a virtualized translation look-aside buffer), and have restricted access to L4-specific features, as we will describe in the following.

3.1 Virtual PowerPC processor

L4 virtualizes cores by mapping each virtual CPU (vCPU) to a dedicated thread. Threads are schedulable entities, and vCPUs are treated equally: They are dispatched regularly from a CPU-local scheduling queue, and they can be moved and load-balanced among individual physical processors through standard L4 mechanisms.

In contrast to recent x86 and PowerPC based architectures, BG/P's cores are based on the widely used embedded 440 architecture, which has no dedicated support to facilitate or accelerate virtualization. However, also in contrast to the x86 architecture, PowerPC is much more virtualization-friendly in the first place: The ISA supports trap-based virtualization, and fixed instruction lengths simplify decoding and emulating sensitive instructions. L4 employs such a trap-and-emulate style virtualization method by compressing PowerPC privilege levels. L4 itself runs in supervisor mode, while the guest kernel and user land both run in user mode (although with different address space IDs, as described in Section 3.2). Guest application code runs undisturbed, but whenever the guest kernel issues a sensitive instruction, the processor causes a trap into L4.

IPC-based virtualization Unlike traditional VMMs, L4 does not emulate all sensitive instructions itself. Unless an instruction is related to the virtual translation look-aside buffer (TLB) or can quickly be handled, like the modification of a guest shadow register state, it hands emulation to the user-level VMM component. Moving the virtualization service out of the kernel makes a fast guest-VMM interaction mechanism a prerequisite for efficient execution. We therefore rely on L4 IPCs to implement the virtualization protocol (i.e. the guest trap – VMM emulation – guest resume cycle). In effect, L4 handles guest traps the same way it handles normal page faults and exceptions, by synthesizing a fault IPC message on behalf of the guest to a designated per-vCPU exception handler (Figure 3).

During a virtualization fault IPC, the trapping guest automatically blocks waiting for a reply message. To facilitate proper decoding and emulation of sensitive instructions, the virtualization IPC contains the vCPU's current execution state such as instruction and stack pointers and general-purpose registers. The reply from the VMM may contain an updated state, which L4 then transparently

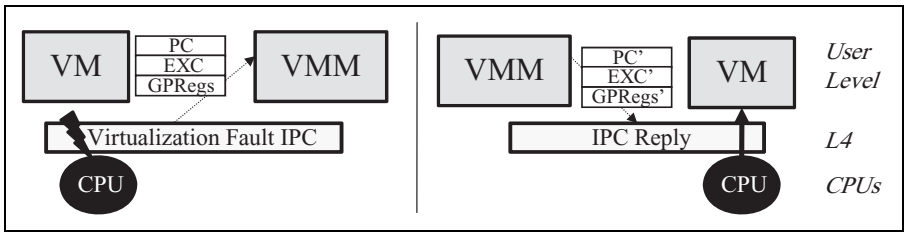

Figure 3. vCPU exits are propagated to the VMM as IPC messages; the user-level VMM responds by sending back a reply IPC message resuming the guest.

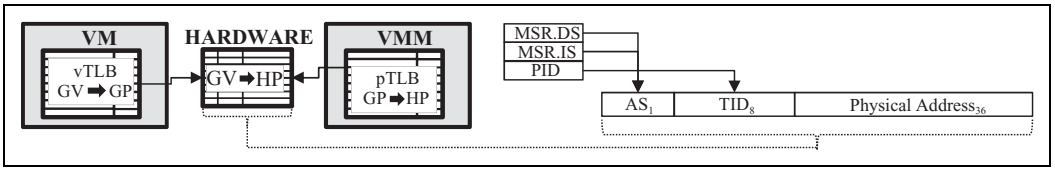

Figure 4. Merging two levels of virtualized address translation into a single-level hardware TLB.

installs into the vCPU's execution frame. To retain the efficient transfer of a guest vCPU state, the kernel allows the VMM to configure the particular state to be transferred independently for each class of faults. For instance, a VMM may choose to always transfer general-purpose registers, but to only transfer a TLB-related guest state on TLB-related faults. L4 also offers a separate system call to inspect and modify all guest states asynchronously from the VMM. That way, we keep the common virtualization fault path fast, while deferring loading of additional guest states to a non-critical path.

3.2 Virtualized memory management

PowerPC 440 avoids die costs for the page table walker and does not dictate any page table format (or even construction). Instead, address translation is based solely on a TLB managed in software (IBM, 2006). On BG/P cores, the TLB has 64 entries managed by the kernel. Each entry maps 32-bit logical addresses to their physical counterparts, and can cover a configurable size ranging from 1 KB to 1 GB. For translation, the TLB further uses an 8-bit process identifier (`PID`) and a 1-bit address space identifier (`AS`), which can be freely set by the kernel, effectively implementing a tagged TLB (Figure 4).

A normal OS only needs to provide a single level of memory translation – from virtual to physical. A VMM, in contrast, must support two such translation levels, from guest-virtual to guest-physical, and from guest-physical to host-physical memory (details og virtualized translations can be found, e.g., in Agesen et al. (2010); KVM Team). As PowerPC 440 hardware only supports a single translation level, the VMM must merge the two levels when inserting translations into the hardware TLB, effectively translating guest-virtual to host-physical addresses. Both L4 and the user-level VMM are involved in providing a two-level memory translation: To provide guest-physical memory, we use L4's existing memory management abstractions based on external pagers (Liedtke, 1995). We then provide a second, in-kernel virtual TLB that provides the additional notion of guest virtual memory. We will describe the L4-specific extensions in the following two paragraphs.

Virtual physical memory Providing guest-physical memory is largely identical to the provisioning of normal virtual memory: L4 treats a guest VM's physical address space like a regular address space and exports establishing of translations within that address-space to a user-level pager (which is, in this case, the user-level VMM). Whenever a guest suffers a TLB miss, L4's miss handler inspects its kernel data structures to find out whether the miss occurred because of a missing guest-physical to host-physical translation. This is the case if the VMM has not yet mapped the physical memory into the VM's guest-physical address space. If so, L4 synthesizes a page fault IPC message to the user-level pager VMM on behalf of the faulting guest, requesting to service the fault.

When the VMM finds the guest-physical page fault to be valid, it responds with a mapping message, which will cause L4 to insert a valid mapping into the TLB and then to resume guest execution. Otherwise, the VMM may terminate the VM for an invalid access to non-existing physical memory or inject a hardware exception. To keep track of a guest's mappings independent of the actual state of the hardware TLB, L4 maintains them with an in-kernel database. Should the user-level VMM revoke a particular mapping, L4 flushes the corresponding database and hardware TLB entries.

Virtual TLB Emulating virtual address translations and the TLB is extremely critical for the overall performance of a VMM. For that reason, most virtualization-enabled hardware platforms have specific support such as recent x86 processors (Bhargava et al., 2008) or embedded PowerPC processors (IBM, 2009). On BG/P, we lack such support and reverted to a software solution using a table shadowing the hardware TLB, which we hence call virtual TLB. In order to further reduce the cost of TLB updates, we employ

a number of heuristics and tracking methods. From a security perspective we only need to ensure that the hardware TLB always contains a subset of the virtual TLB.

L4 provides a virtual TLB, which the guest has access to via normal (but trapped) hardware instructions for TLB management. While the management of guest-physical memory involves the user-level VMM, our solution for guest-virtual memory is L4-internal: Whenever the guest kernel accesses the virtual TLB, L4's internal instruction emulator stores those entries into a per-VM virtual TLB data structure. On a hardware TLB miss, L4's miss handler parses that data structure to find out whether the guest has inserted a valid TLB mapping into its virtual TLB for the given fault address. If not, it injects a TLB miss fault into the guest VM to have the miss handled by the guest kernel. If the virtual TLB indeed contains a valid entry, L4 checks its mapping database to find out whether the miss occured at the second stage, from guest-physical to host-physical. If that translation is valid as well, L4 inserts the resulting guest-virtual to host-physical mapping into the hardware TLB and resumes the VM; if it turns out to be missing, L4 synthesizes a page fault IPC to the VMM, as discussed in the previous paragraph.

Virtual address space protection Finally, a VMM must virtualize not only the translation engine of the TLB but also its protection features. Again, the virtualization logically requires two levels, allowing the guest to use the virtual TLB's protection bits and identifiers in the same manner as on native hardware, but, at the second level, also permitting L4 and its user-level address spaces to shield their data from being accessed by guest kernel and applications. The TLB of the PowerPC 440 is very useful. The 440 can hold up to 256 address space mappings in the TLB (via the `TID` field, see Figure 4). The particular mapping is chosen through a processor register (`TID`). Address space translation 0 is always accessible independent of which particular mapping is active. The 440 additionally features a 1-bit translation space, with the active translation space being selected via processor register bits, one for instruction fetches and one for data fetches (`MSR.IS`, `MSR.DS`).

To facilitate trap-and-emulate virtualization, both guest kernel and applications run in user mode. L4 puts the guest kernel and user into the second translation space, and reserves the first translation space for itself, the VMM and native L4 applications. The processor automatically switches to the first translation space when an interrupt or trap occurs, directly entering L4 with trap and interrupt handlers in place. To read guest memory when decoding sensitive instructions, L4 temporarily switches the translation space for data fetches, while retaining the space for instructions. Altogether, our solution allows the guest to receive a completely empty address space, but on any exception, the processor switches to a hypervisor-owned address space. That way, we keep virtualization address spaces clean without the need for ring compression as done on x86 systems lacking hardware virtualization (Uhlig et al., 2005).

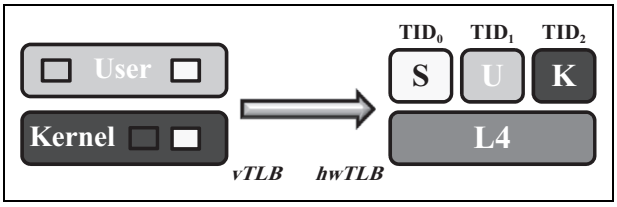

Figure 5. Virtualized TLB Protection. vTLB protection bits are mapped to `TID` in the hardware TLB, with `TID=0` for shared pages.

Our ultimate goal is to reduce the number of TLB flushes to enforce protection on user-to-kernel switches. We observed the following usage scenario for standard OSs (e.g. Linux) and implemented our algorithm to mimic the behavior: Common OSs map application code and data as user and kernel accessible, while kernel code and data is only accessible in privileged mode. We strive to make the transition from user to kernel and back fast to achieve good system call performance; we therefore disable the address space mappings completely and flush the hardware TLB on each guest address space switch (Figure 5). We then use address space 0 for mappings that are accessible to guest user and guest kernel. We use address space 1 for mappings that are only guest-user accessible and address space 2 for mappings that are only guest-kernel accessible. A privilege level switch from guest-user to guest-kernel mode therefore solely requires updating the address space identifier from 1 (user-mode mappings) to 2 (kernel-mode mappings).

Our virtual TLB effectively compresses guest user/kernel protection bits into address space identifiers; as a result, it requires hardware TLB entries to be flushed whenever the guest kernel switches guest application address spaces. Also, our scheme requires that TLB entries are flushed during world switches between different guests. It does not require, however, any TLB flushes during guest system calls or other switches from guest user to kernel, or during virtualization traps and resumes within the same VM. Thus, we optimize for frequent kernel/user and kernel/VMM switches rather than for address space or world switches, as the former occur more frequently.

3.3 Interrupt virtualization

BG/P provides a custom interrupt controller called the Blue Gene Interrupt Controller (BIC), which gathers and delivers device signals to the cores as interrupts or machine check exceptions. The BIC supports a large number (several hundreds) of interrupts, which are organized in groups and have different types for prioritization and routing purposes. The BIC supports steering interrupts among different cores as well as core-to-core interrupt delivery.

The original L4 version provided support for user-level interrupt management, mapping interrupt messages and acknowledgments onto IPC. L4 further permits user software to migrate interrupts to different cores. Our user-level VMM uses those L4 interrupt features to receive and acknowledge interrupts for BG/P devices. To inject virtual

```
void vcpu_t::emulate (ppc_instr_t opcode, word_t gva, paddr_t gpa)
{
  // fetch guest GP registers from L4 IPC message
  L4_GPRegsCtrlXferItem_t gpregs;
  L4_MsgGetGPRegsCtrlXferItem (&msg, mr, gpregs);

  // Decode instruction
  switch (opcode.primary()) {
  case 31:
    switch (opcode.secondary()) {
    case 259:   // mfdcrx
      // invoke dcr/device specific emulation
      emulate_mfdcr (Dcr (*gpr (gpregs, opcode.ra())), gpr (gpregs, opcode.rt()));
      break;
    }
  }
}
```

Figure 6. The process of emulating device accesses in the VMM. The fixed opcode lengths of PowerPC drastically simplify decoding of instructions.

interrupts into the guest, the VMM modifies the guest vCPU state accordingly, either using L4's state modification system call (Section 3.1), or by piggybacking the state update onto a virtualization fault reply, in case the guest VM is already waiting for the VMM when the interrupt occurs.

4 User-level VMM

Our user-level VMM component runs as a native L4 program, and provides the virtualization service based on L4's core abstractions. It can be described as an interface layer that translates virtualization API invocations (i.e. sensitive instructions) into API invocations of the underlying L4 architecture. As described, L4 facilitates virtualization by means of virtualization fault IPC. The user-level VMM mainly consists of a server executing an IPC loop, waiting for any incoming IPC message from a faulting guest VM. Upon reception, it retrieves the VM register context that L4 has sent along, emulates the sensitive instruction accordingly, and finally responds with a reply IPC containing an updated vCPU state such as result registers of the given sensitive instruction and an incremented program counter. Before resuming the VM, L4 installs the updated context transparently into the VM, while the VMM waits for the next message to arrive.

4.1 Emulating sensitive instructions

Our user-level VMM largely resembles other typical VMMs such as VMware or Xen: It contains a virtual CPU object and a map translating guest physical memory pages into memory pages owned by the VMM. To emulate sensitive instructions upon a virtualization fault IPC, the VMM decodes the instruction and its parameters based on the program counter pointer and general-purpose register file of the guest VM, which are stored within the IPC message that was sent from L4 on behalf of the trapping VM. For convenience, L4 also passes on the *value* of the program counter, that is, the trapping instruction. In comparison to x86 processors, which have variable-sized instructions of lengths up to 15 bytes, fetching and decoding sensitive instructions on embedded PowerPC are rather trivial tasks, as instructions have a fixed size of 32 bits on PowerPC. The code listing in Figure 6 illustrates the emulation process by example of a move from device control register (`mfdcrx`) instruction loading the value of a device register into a general-purpose register.

4.2 Virtual physical memory

To the user-level VMM, paging a guest with virtualized physical memory is similar to regular user-level paging in L4 systems (Härtig et al., 1997): Whenever the guest suffers a physical TLB miss, L4 sends a page fault IPC containing the faulting instruction and address and other (virtual) TLB states necessary to service the fault. In its present implementation, the VMM organizes guest-physical memory in linear segments. Thus, when handling a fault, the VMM checks whether the accessed guest-physical address is within the segment limits. If so, it responds with a mapping IPC message that causes L4 to insert the corresponding mapping into its database and into the hardware TLB.

4.3 Device virtualization

Besides virtualization of BG/P cores, the main task of the user-level VMM is to virtualize BG/P hardware devices. L4 traps and propagates sensitive device instructions such as moves to or from system registers (`mfdcr`, `mtdcr`), as well as generic load/store instructions to sensitive device memory regions. It is up to the VMM to back those transactions with appropriate device models and state machines that give the guest the illusion of real devices, and to multiplex them onto actual physical hardware shared among all guests. The VMM currently provides virtual models for

```
word_t vcn_chan_t::gpr_store (Reg reg, word_t *gpr)
{
 switch (reg) {
  case PIH:
   if (ihead_cnt() == fifo_head_size)
    cn->pixf |= vcn_t::PIX_HF00 >> channel;
   else {
    ififo_head[ihead_add++ % fifo_head_size] = *gpr;
    send_packet();
   }
   cn->raise_irqs();
   break;
 }
}

word_t vcn_chan_t::fpr_store (Reg reg, unsigned fpr)
{
 switch (reg) {
  case PID:
   if (idata_cnt() == fifo_dsize)
    cn->pixf |= vcn_t::PIX_PF00 >> channel;
   else {
    fpu_t::read (fpr,
        &ififo_data[idata_add++ % fifo_dsize]);
    // Clear watermark exception
    if (idata_cnt() / 16 > iwatermark)
     cn->pixf &= ~(vcn_t::PIX_WM0 >> channel);
    send_packet();
   }
   cn->raise_irqs();
 }
}
```

Figure 7. Storing the contents of a general-purpose or floating-point register into the respective device registers of the virtualized collective device.

the BIC and for the collective and the torus network devices. Emulation of the BIC is a rather straightforward task: The VMM intercepts all accesses to the memory-mapped BIC device and emulates them using L4's mechanisms for external interrupt handling and event injection. The following paragraphs detail the emulation of the collective and torus network.

Collective network BG/P's collective network is an over-connected binary tree that spans the whole installation. The collective is a one-to-all medium for broadcast or reduction operations, with support for node-specific filtering and a complex routing scheme that can sub-partition the network to increase the total bandwidth. The collective link bandwidth is 6.8 Gbit/s; hardware latencies are below 6 μs for a 72-rack system (IBM Blue Gene team, 2008). Software transmits data over the collective network via packets injected into two memory-mapped virtual channels. Each packet consists of a header and 16 128-bit data words. The packet header is written and read via general-purpose registers, using normal load and store instructions to and from device memory; the packet data is written and read through the floating-point unit.

Our VMM provides a fully virtualized version of BG/P's collective network device. The VMM leaves device memory unmapped, so that each device register access leads to a virtualization trap. The VMM furthermore emulates device control registers (DCRs) used to configure the collective network device. The corresponding instructions are sensitive and directly trap to L4 and the VMM. For emulation, the VMM provides a per-VM shadow collective network interface model that contains virtual DCRs and, per channel, virtual injection and reception FIFOs with a virtual header and 16 virtual FPU words per packet, as the code snippets in Figure 7 illustrate.

The VMM registers itself to L4 as an interrupt handler for all collective network device interrupts, which will cause L4 to emit an interrupt IPC message to the VMM whenever the physical device fires one of its hardware interrupts. Whenever the guest causes a packet to be sent on its virtual collective network interface, the VMM loads the corresponding

```
void vcn_chan_t::send_packet()
{
 // Need at least one header and
 // one payload to send a packet
 if (ihead_cnt() && idata_cnt() / 16) {

  // Use corresponding phys channel
  pcn_t::cn.chan[channel].
    send_packet (ififo_head[ihead_rem % fifo_hsize],
                 &ififo_data[idata_rem % fifo_dsize]);
  ihead_rem += 1;
  idata_rem += 16;

  // Flag watermark exception
  if (idata_cnt() / 16 <= iwatermark)
   cn->pixf |= vcn_t::PIX_WM0 >> channel;
 }
}
```

Figure 8. Sending a packet via the physical collective device; it is called by the VMM whenever the guest issues a send operation on the virtualized collective device.

virtual registers into the physical collective network device (Figure 8).

Receiving data is slightly more complex: Whenever the VMM receives an interrupt message from L4, it reads the packet header and data from the physical device into a private buffer, and then delivers a virtual interrupt to the corresponding guest VM. Subsequent VM accesses to the packet header and data are then served out of the private buffer into the VM's general-purpose or floating-point registers. Since copying packets induces substantial a software overhead for the high-performance collective network path (Section 6), we are also considering optimized virtual packet handling by means of device pass-through and/or para-virtualization techniques.

Torus BG/P's torus network is the most important data transport network with respect to bisectional bandwidth, latency, and software overhead (Adiga et al., 2005). Each compute node is part of the 3D torus network spanning the whole installation. On each node, the torus device has input and output links for each of its six neighbors, each with a bandwidth of 3.4 Gbit/s, for a total node bandwidth of 40.8 Gbit/s. Worst-case end-to-end latency in a 64 k server system is below 5 μs. Nodes act as forwarding routers without software intervention.

The torus provides two transmission interfaces, a regular buffer-based interface and one based on remote direct memory access (rDMA). For the latter, the torus DMA engine provides an advanced put/get-based interface to read or write segments of memory from or to remote nodes, based on memory descriptors inserted into its injection and reception FIFOs. Each memory descriptor denotes a contiguous region in physical memory on the local node. To facilitate rDMA transactions (e.g. a direct-put operation copying data directly to a remote node's memory),

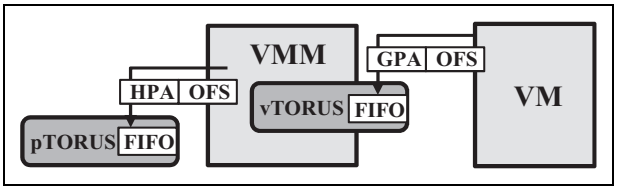

Figure 9. Virtualized torus interconnect. The VMM passes through guest VM descriptors, translating addresses from guest- to host-physical.

software identifies the corresponding remote memory segments via a selector/offset tuple that corresponds to a descriptor in the receive FIFO on the remote node (Figure 9). To transmit data via the torus, software inserts one or more packets into a torus injection FIFO, with the packet specifying the destination using its X,Y,Z coordinates. For non-DMA transfer, the payload is embedded in the packet; for DMA transfers (local as well as remote), the corresponding sender and receiver memory segment descriptors and selectors are instead appended to the packet.

As with the collective network, our VMM provides a virtualized version of Blue Gene's torus device, and traps and emulates all accesses to torus device registers. Again, DCR instructions directly trap into L4, while device memory accesses trap by means of invalid host TLB entries. Again, the VMM registers itself for physical torus interrupts and delivers them to the guests as needed.

However, in contrast to our virtual version of the collective network device, our virtual torus device only holds the DMA descriptor registers, but does not copy the actual data around during DMA send or receive operations. Instead, it passes on guest VM memory segment descriptors from virtual to physical FIFOs, merely validating that they

```
torus_remote_get(to_coordinates, to_counter_id, to_offset,
                 from_coordinates, from_fifo_id, from_ctr_id, from_offset,
                 len);
if (poll)
  bg_torus_poll_rx(len);
```

Figure 10. Issuing a remote get operation on the torus device and polling for the result.

reference valid guest-physical memory before translating them into host-physical addresses. As the VMM currently uses simple linear segmentation (see Section 4.2), that translation is merely an offset operation and the descriptors in physical FIFOs always reference valid guest memory.

In contrast to general-purpose virtualization environments, which typically provide virtualized Ethernet devices including virtual MAC addresses and a virtual switch architecture, our supercomputer VMM presently does not provide extra naming or multiplexing of multiple virtual torus devices. Instead, it preserves the naming of the real world and maps virtual torus coordinates and channel identifiers idempotently to their physical counterparts. As a result, our VMM does not fully emulate the torus network, but merely provides safe and *partitioned* access to the physical torus network for individual VMs. As BG/P's torus features four independent DMA send and receive groups, our VMM can supply up to four VMs with a different DMA channel holding all its descriptors, without having to multiplex descriptors from different VMs onto a single FIFO.

At present, our VMM intercepts all accesses to the virtual DMA groups and multiplexes them among the physical ones. However, since DMA groups are located on different physical pages and thus have different TLB entries, we plan, for future versions, to directly map torus channels into guest applications, effectively allowing them to bypass the guest OS *and hypervisor*. While such a pass-through approach will require a para-virtual torus driver (to translate guest-physical to host-physical addresses), it can still be made safe without requiring interception, as BG/P's torus DMA engine supports a set of range check registers that enable containment of valid DMA addresses into the guest's allowed allotment of physical memory.

5 Native application support

The field of research on native application frameworks for μ-kernels has been rich (Härtig et al., 2005; Kuz et al., 2007), and has even been explored for HPC systems (Lange et al., 2010). To a certain extent, the traditional CNK approach for BG/P can also be termed a μ-kernel effort, since it also strives to provide a low-footprint core environment for running HPC applications. In the following section, we describe how L4 facilitates construction of a user-level environment that allows the running of HPC applications as well as controlling their resource demands and allocations such that they perform well on high-performance hardware. We do so by means of an example: we construct a core HPC communication library that can be used to perform rDMA-based data transfers at the user level. We then integrate our library into a sample client-server application running natively atop L4.

Unlike typical HPC lightweight kernels, such as CNK or Palacios, L4 does not make any effort to preserve Linux or POSIX compatibility as a first-class abstraction; rather, it provides a set of primitives that allow constructing arbitrary OS personalities atop (amongst others, a VMM preserving POSIX compatibility at the guest OS layer). The whole design of L4 is centered around the idea of a single, generic, and efficient local IPC communication primitive.

5.1 A native torus communication library

Our torus communication library is a protocol library that allows a communication partner to read and write portions of the memory of another partner via remote DMA. As described in Section 4.3, the torus supports different rDMA operations. The two most important for our library are: (a) the `remote-get` function copying memory from a remote node's memory into the local (or a different) node's memory; and (b) the `direct-put` operation copying data directly from a local to a remote node's memory.

The torus hardware features multiple injection and reception FIFOs that can be used to transfer data; for rDMA operations, the torus additionally provides four different DMA groups (one per core) that allow offloading of the marshaling and packetizing of memory segments to hardware. To denote memory segments, the torus hardware introduces counters, which are physically contiguous memory segments between two associated physical addresses (`base` and `limit`); a counter additionally features a counter variable that is incremented by the data size whenever data has been transferred via that counter, allowing software to keep track of the data transmissions. Our library directly exposes those functions as its main interface; in addition, the library provides and manages the contiguous memory segments necessary for the actual data transfer. Thus, a typical remote-get operation in our library looks like the one depicted in Figure 10.

In this example, the invoker instructs the torus to fetch some data portion from the node specified by `from_coordinates` and, on that node, within the memory segment specified by counter `from_ctr_id` at offset `from_offset`. It further instructs the torus to copy that data portion to node `to_coordinates`, and again, on that node, into the memory segment specified by

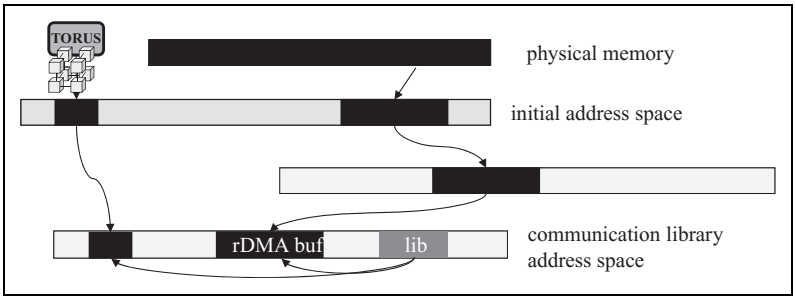

Figure 11. Mapping physically contiguous addresses in native L4 address spaces.

counter `to_ctr_id` at offset `to_offset`. The destination node usually is (but does not have to be) the invoker's own node. Note that the actual memory layout on the nodes stays opaque to the user of the torus; only counter identifiers and relative offsets need to be published to transmit data. As also illustrated by this example, the transfer can be monitored for completion by polling the receive channel. In order to overlap torus I/O with other computation, the torus can alternatively be configured to generate an interrupt after a certain number of bytes has been transferred.

5.2 Memory management

As with most DMA engines, the torus rDMA engines operates on physical addresses: As a result, memory segments used to communicate data between nodes must be contiguous physical regions.

For our communication library, we therefore employ a two-level memory provisioning and management scheme: we first map coarse-grain memory chunks into the address space where the communication library lives, using standard L4 recursive mapping primitives. Within that address space, we then use a standard `malloc/free/realloc` implementation to allow library users to further sub-allocate segments usable for data communication:

User-level allocation of physically contiguous memory L4's main memory management primitive is a mapping operation that allows the invoking thread to transfer its own permissions to access memory to another address space. The operations leverages L4's IPC mechanism presented beforehand; that is, an IPC can contain special message items denoting a memory mapping from the source to the destination address space. For revocation of memory rights, L4 provides a separate kernel primitive that does not require explicit consent from any of the existing right receivers. This recursive mapping starts with a root-level address-space called sigma0 (Liedtke, 1995), which 'owns' all physical memory in the system, that is, which has all physical memory mapped idempotently in its own address space.

We use standard L4 mapping semantics to bring memory into the native applications using our torus communication library. We additionally ensure, by means of a user protocol, that (a) memory reserved for torus communication is physically contiguous and the translation from virtual to physical is known to the library, and (b) that memory will not be revoked from the library address space without notice. To ensure the former, we map each chunk as a whole and communicate the physical base addresses from the root pager down the mapping chain to the library addresses space, with each pager hierarchy performing the translation of its own source to destination mapping (Figure 11). To ensure the latter, we currently mark the mappings globally unrevokeable, disallowing each pager to ever revoke any of those mappings until the application itself releases the torus memory resources. In other words, we prevent the memory used for rDMA transactions from being paged out, while still allowing deallocation at application level.

With the architecture of the embedded PowerPC 440 architecture, it is up to the L4 kernel to maintain active translations in the TLB. As a result, a memory mapping from one to the other address space that is never revoked will cause L4 to store a permanent mapping in its internal mapping database; however, the translation may still be evicted from the TLB occasionally due to pressure (and L4's TLB handler has to re-insert them on the next access). While the torus rDMA engine operates on physical memory and bypasses the TLB, our software library may suffer performance drawbacks when reading or writing communication memory regions if they are not present in the TLB. To avoid the resulting jitter and non-deterministic performance variations, we propose to add special flags to the L4 mapping primitive indicating L4 should mark the translations as pinned in the TLB, such that they never fault. Obviously, such pinned mappings constitute a scarce resource and their use needs to be managed. While we currently rely an a cooperative scheme to avoid exhaustion, we refer to the literature for more elaborate scheduling and/or pinning of un-trusted memory (Liedtke et al., 1999).

Fine-grain allocation within physically contiguous memory Within each address space where our communication library is running, we sub-divide coarse-grain memory chunks into smaller pieces by means of a slab memory allocator (Lea). The memory allocator allows the sub-dividing of a memory chunk into memory spaces called `mspace`; whenever the torus library needs to set up a new receive or send memory segment for use by the torus, it creates a new `mspace`; subsequent rDMA transactions (such as `torus_remote_get` described above) can then conveniently use `malloc` and `free` to acquire buffer space usable for

```
int torus_alloc_rx(torus_t *torus, torus_channel_t *chan, size_t buf_size)
{
    long long pbuf;
    int ctr_grp, ctr_idx;

    // allocate a DMAable mspace
    chan->dma_space = create_mspace(buf_size);
    chan->buf = mpace_base(chan->dma_space);
    chan->buf_size = mspace_size(chan->dma_space);

    // allocate a torus receive counter
    chan->ctr = alloc_rx_counter();

    // enable torus receive counter
    ctr_grp = ctr / BGP_TORUS_COUNTERS;
    ctr_idx = ctr % BGP_TORUS_COUNTERS;

    // enable bits are grouped into fields of 32-bit size
    // find bit numbered ctr_idx
    torus->dma[ctr_grp].rcv.counter_enable[ctr_idx / 32] =
        (0x80000000 >> (ctr_idx % 32));
    chan->rx.ctr = &torus->dma[ctr_grp].rcv.counter[ctr_idx];

    // install mspace in torus using its physical address
    pbuf  = dma_vmem_to_pmem(chan->buf);
    chan->rx.ctr->base = pbuf >> 4;
    chan->rx.ctr->counter = 0xffffffff;

    chan->rx.received = 0;

    return 0;
}
```

Figure 12. Native library function for allocating a torus receive counter.

transmitting data via the torus. Figure 12 illustrates the memory management by describing the function that sets up a new receive counter and memory segment for use by the torus.

6 Initial evaluation

Our approach is in a prototypical stage, and we have not yet optimized any of the frequently executed trap-and-emulate paths. For evaluation, we thus focused mostly on functionality rather than performance. Nevertheless, we have run some initial performance benchmarks to find out whether our approach is generally viable and where the most important bottlenecks and possibilities for optimization reside.

Guest OS support Our L4-based VMM generally supports the running of arbitrary guest OSs on BG/P, such as CNK (Giampapa et al., 2010) or ZeptoOS (Beckman et al., 2008). With respect to the implementation of the virtualized TLB, L4 is currently limited in that it reserves one of the two translation spaces. We have verified our implementation with Kittyhawk Linux, a BG/P version of the Linux Kernel (Appavoo et al., 2009) with support for BG/P's hardware devices. It also provides an overlay that maps standard Linux Ethernet communication onto Blue Gene's high-speed collective and torus interconnects (Appavoo et al., 2010). Our VMM allows one or more instances of Kittyhawk Linux to run in a VM. Kittyhawk Linux runs unmodified, that is, the same Kittyhawk Linux binary that runs on BG/P also runs on top of our VMM. We currently support uni-processor guests only; however, as L4 itself supports multi-processing, individual guest vCPUs can be scheduled on each of the four physical cores of the Blue Gene node.

Initial benchmark results For initial evaluation, we ran three experiments. In the first experiment, we compiled a small source-code project (about 1000 lines of code) under virtualized Kittyhawk Linux. We then used a debug build of L4 with an internal event-tracing facility to find out frequently executed VM-related code paths. In this configuration, compilation took about 126 s compared to 3 s when running on native Kittyhawk Linux. Table 1 lists the results.

We note that the number of IPCs – that is, the number of VM exits that involve the user-level VMM – is relatively low, meaning that L4 handles most guest exits internally. Also the experiment shows a high number of TLB misses and TLB-related instructions, indicating that the virtualized memory subsystem is a bottleneck in our implementation.

In the second experiment, we measured Ethernet network throughput and latency between two compute

Table 1. Execution frequency of VMM-related L4 code paths for a compilation job in a VM.

Trace Point	Count	Trace Point	Count	Trace Point	Count
SYSCALL_IPC	106 K	ITLB_MISS	4756 K	EMUL_MFMSR	228 K
EXCEPT_DECR	66 K	EMUL_RFI	5021 K	EMUL_WRTEEI	403 K
EMUL_MTMSR	102 K	DTLB_MISS	6514 K	EMUL_MFSPR	29964 K
EMUL_WRTEE	117 K	EMUL_TLBWE	14745 K	EMUL_MTSPR	30036 K

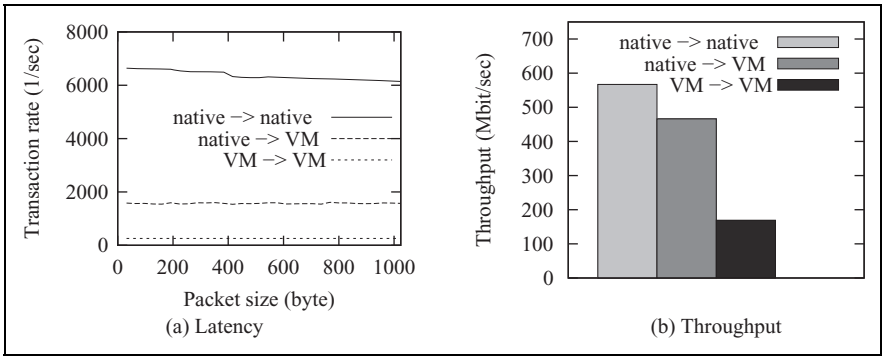

Figure 13. Performance of torus-routed virtualized Ethernet networks.

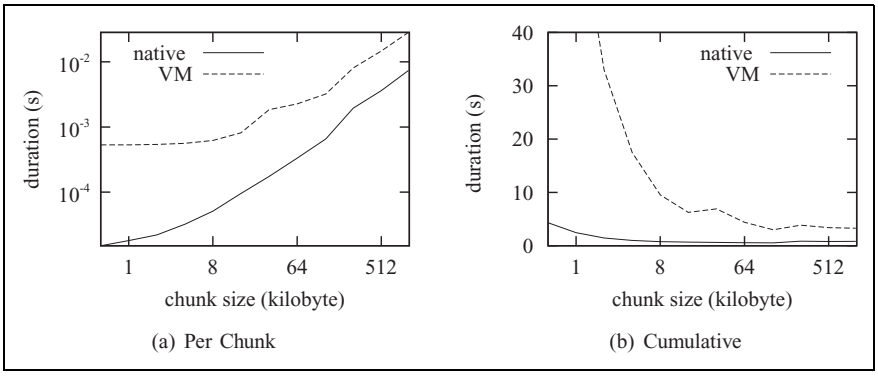

Figure 14. Performance of our torus communication library running natively atop L4 versus running virtualized in a Linux guest VM.

nodes. Packets are delivered to the torus interconnect, by means of Kittyhawk Linux's Ethernet driver module. For comparison, we ran the experiments for both a native and virtualized Kittyhawk Linux each running on a compute node. For benchmarking we used `netperf`'s TCP stream test for throughput and the TCP request/response test for latency measurements (Netperf Team). Figure 13 shows the results.

Our virtualization layer poses a significant overhead on the Ethernet network performance, which is already less than the actual performance that the torus and collective hardware can deliver (Appavoo et al. 2010). We are confident, however, that optimizations such as para-virtual device support can render virtualization substantially more efficient. A recent study reports that VMware's engineers faced, and eventually addressed, similarly dismal performance with prototypical versions of their VMM (Agesen et al., 2010).

Native communication library Finally, in the third experiment, we compared the throughput of our native communication library against a version of our library running within a Linux guest VM. To that end, we ported our communication library to Kittyhawk Linux, allowing it to run within a normal Linux application. For measurements, we developed a simple benchmark that repeatedly fetches fixed-sized chunks of memory from a remote compute node via the torus, using our communication library. We ran the benchmark application natively on L4 and virtualized in a guest Linux application, and compared the respective times necessary to fetch the data. In contrast to the second experiment, there is no Ethernet virtualization layer involved, since the library interfaces directly with the torus (or virtual torus) to transmit data. Figure 14 shows the results for different chunk sizes. The first plot shows the transfer time for a single chunk with a logarithmic scale, while the second plot shows the total time needed to transfer 100 MB of data.

As can be seen, native rDMA performance is significantly higher than with the virtualized torus implementation, clearly showing the potential of L4's support for

the native applications stack. For small payloads (1 KB), the native transmission outperforms the virtualized by far. This can be attributed to the high performance of the interconnect, the relatively low speed of the cores, and the resulting high processing costs for VM exits and entries that are necessary for each transmit of a chunk. For larger chunk sizes, the virtualization overhead becomes less dominant.

Altogether, we conclude from the preliminary evaluation and our own experiences: that our L4-based VMM is a promising approach for successfully deploying, running, and using virtualized OSs and native applications on BG/P; that virtualization performance of our prototype lags substantially compared to more mature or commercial virtualization efforts from the commodity server space, warranting further exploration and optimization; and that native application environments, which execute directly atop the micro-kernel and have direct access to hardware facilities, are a key requirement towards fully exploiting the capabilities of a supercomputer and its cutting-edge, high-performance interconnects.

7 Related work

There exists a plethora of VMM efforts, including Xen (Pratt et al., 2005), VMware (Agesen et al., 2010), and, directly related, micro-hypervisor-based systems (Heiser and Leslie, 2010; Steinberg and Kauer, 2010: Härtig et al., 1997); those approaches mostly address the embedded or server spaces rather than HPC space. The studies in Engelmann et al.(2007)] and [Mergen et al.(2006)] identified virtualization as a system-level alternative to address the development challenges of HPC systems. Gavrilovska et al. explored virtualized HPC for x86/InfiniBand-based hardware (Gavrilovska et al., 2007); PROSE explored a partitioning hypervisor architecture for PowerPC- and x86-based HPC systems (Van Hensbergen, 2006). However, both approaches focus mostly on hypervisor infrastructure rather than on decomposed OS designs or on support for native applications. Finally, there exists a port of the KVM monitor to PowerPC Book E cores (KVM Team); although designed for embedded systems, it shares some implementation details with our PowerPC version of L4 and the VMM.

Arguably the work most closely related to our approach is Palacios and Kitten (Lange et al., 2010), a lightweight kernel/VMM combination striving to achieve high performance and scalability on HPC machines. Palacios and Kitten are being developed for x86-based HPC systems rather than for a highly-specialized supercomputer platform such as BG/P. Palacios runs as a module extension within the Kitten kernel. Like L4, Kitten also provides a native environment that can be used to develop customizations. Also, and much akin to the micro-kernel paradigm, Palacios and Kitten comprise a fairly small kernel code base of around 120 K lines of code (Lange et al., 2010), which is comparable to the around 60 K lines of our L4 micro-kernel version (note, however, that L4Ka::Pistachio supports x86, x86_64, and PowerPC, while Palacios/Kitten only supports x86 and x86_64).

For effective virtualization of HPC networks, Kitten and Palacios introduce a scheme that relies on guest cooperation, in order to preserve the high performance of the interconnect (Lange et al., 2011). In contrast, we currently fully intercept guest device accesses, which negatively affects network performance but preserves exact device semantics and allows using unmodified interconnect drivers in a VM. We regard this as a minor difference mostly stemming from our early research aims to support unmodified guest code, and are confident that our VMM can be adapted in a straightforward manner to support para-virtualized, but more efficient, device drivers in the guest.

The most notable conceptual difference between Palacios/Kitten and our L4-based VMM approach is that Palacios runs as a kernel module in Kitten's privileged domain, whereas our VMM runs completely decomposed and deprivileged as a user-level process. We argue that decomposing the VMM has benefits to system structure, stability, extensibility, and fault isolation, while the purported negative effects on overhead can be mitigated by careful design and implementation. Recent research substantiates our claim, showing that such a decomposed thin virtualization layer can indeed be built with negligible performance overhead (Steinberg and Kauer, 2010).

Examples of traditional lightweight kernel approaches for supercomputers are CNK (Moreira et al., 2006; Giampapa et al., 2010) and Sandia's Catamount (Kelly and Brightwell, 2005). Our approach strives to enhance such lightweight kernel approaches in that it provides the ability to virtualize guest OSs. Finally, research has explored whether a more fully fledged OS such as Plan9 (Minnich and McKie, 2009) or Linux (Appavoo et al., 2009; Beckman et al., 2008; Kaplan, 2006) may be a more viable alternative than lightweight kernels for supercomputers. Our approach complements those efforts with the alternative idea of a decomposed OS architecture with support for virtualization.

8 Conclusion

In this paper, we have presented a light-weight operating system and virtualization architecture for the Blue Gene/P Supercomputer. μ-kernel OS architecture for BG/P. Our architecture consists of a virtualization-capable μ-kernel and a user-level VMM component running atop. The μ-kernel also supports running applications natively. Our L4-based prototype successfully virtualizes Kittyhawk Linux with support for virtualized collective and torus network devices. Our first experiences and experiments show that our VMM still takes a substantial performance hit. However, we believe that, pending optimization, our approach poses an interesting OS alternative for supercomputers, providing the convenience of a fully featured commodity OS software stack, while also promising to satisfy the need for low latency and scalability of a HPC system.

Besides performance improvements of the VMM, we consider the main area for future work to be applications that make use of our approach: we believe there is a broad range of applications and workloads – whether traditional supercomputer applications such as large-scale, MPI-style simulations or performance and scalability demanding tasks from the commodity systems world such as big data analytics or stream-processing tools – that could heavily benefit from our hybrid architecture, since it enables HPC programmers to employ familiar OS abstractions inside a VM to quickly develop a general HPC solution and then gradually roll out scalable, low-latency services running natively alongside. In particular, future work has to be done to explore such hybrid programming models on the native side of L4, where our existing low-level native interface can serve as a starting point.

Funding

This work used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the US Department of Energy [contract number DE-AC02-06CH11357].

Conflict of interest statement

None declared.

Acknowledgements

The infrastructure presented herein is part of the the projects L4Ka and Kittyhawk. The software is open source and can be downloaded from http://l4ka.org and http://kittyhawk.bu.edu.

References

Adiga NR, Blumrich MA, Chen D, Coteus P, Gara A, Giampapa ME, Heidelberger P, Singh S, Steinmacher-Burow BD, Takken T, Tsao M and Vranas P (2005) Blue Gene/L torus interconnection network. *IBM Journal of Research and Development* 49(2/3): 265–276.

Agesen O, Garthwaite A, Sheldon J and Subrahmanyam P (2010) The evolution of an x86 virtual machine monitor. *ACM Operating Systems Review* 44(4): 3–18.

Appavoo J, Uhlig V, Stoess J, Waterland A, Rosenburg B, Wisniewski R, Silva DD, van Hensbergen E and Steinberg U (2010) Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, Chicago, IL, USA, pp. 385–394.

Appavoo J, Uhlig V, Waterland A, Rosenburg B, Silva DD and Moreira JE (2009) Kittyhawk: Enabling cooperation and competition in a global, shared computational system. *IBM Journal of Research and Development* 53(4): 1–15.

Beckman P, Iskra K, Yoshii K, Coghlan S and Nataraj A (2008) Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing* 11(1): 3–16.

Bhargava R, Serebrin B, Spadini F and Manne S (2008) Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on architectural support for programming languages and operating systems*, Seattle, WA, USA, March 2008, pp. 48–59.

Engelmann C, Scott S, Ong H, Vallée G and Naughton T (2007) Configurable virtualized system environments for high performance computing. In *1st workshop on system-level virtualization for high performance computing*, Lisbon, Portugal, March 2007.

Gavrilovska A, Kumar S, Raj H, Schwan K, Gupta V, Nathuji R, Niranjan R, Ranadive A and Saraiya P (2007) High-performance hypervisor architectures: Virtualization in HPC systems. In *1st workshop on system-level virtualization for high performance computing*, Lisbon, Portugal, March 2007.

Giampapa M, Gooding T, Inglett T and Wisniewski RW (2010) Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of the 2010 international conference on supercomputing*, New Orleans, LA, USA, November 2010, pp. 1–10.

Härtig H, Hohmuth M, Feske N, Helmuth C, Lackorzynski A, Mehnert F and Peter M (2005) The Nizza secure-system architecture. In *Proceedings of the 1st international conference on collaborative computing: Networking, applications and worksharing*, San Jose, CA, USA, December 2005.

Härtig H, Hohmuth M, Liedtke J and Schönberg S (1997) The performance of u-kernel based systems. In *Proceedings of the 16th symposium on operating system principles*, Saint Malo, France, October 1997, pp. 66–77.

Heiser G and Leslie B (2010) The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific workshop on systems*, August 2010, pp. 19–24.

IBM Power ISA Version 2.03. IBM Corporation.

IBM (2009) *IBM Power ISA Version 2.06*. IBM Corporation.

IBM Blue Gene team (2008) Overview of the IBM Blue Gene/P Proj ect. *IBM Journal of Research and Development* 52(1/2): 199–220.

Kaplan LS (2006) Lightweight Linux for High-performance Computing. Linux World.com, http://www.linuxworld.com/news/2006/120406-lightweight-linux.html.

Kelly SM and Brightwell R (2005) Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray user group annual technical conference*, Albuquerque, NM, USA, May 2005.

Kuz I, Liu Y, Gorton I and Heiser G (2007) Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software* 80(5): 687–699.

KVM Team. KVM for PowerPC. http://www.linux-kvm.org/page/PowerPC/. Accessed Aug 2010.

L4 Development Team (2009) *L4 X.2 Reference Manual*. University of Karlsruhe, Germany, May 2009.

Lange JR, Pedretti K, Dinda P, Bridges PG, Bae C, Soltero P and Merritt A (2011) Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, Newport Beach, CA, USA, March 2011, pp. 169–180.

Lange JR, Pedretti KT, Hudson T, Dinda PA, Cui Z, Xia L, Bridges PG, Gocke A, Jaconette S, Levenhagen M and Brightwell R (2010) Palacios and Kitten: New high performance

operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE international symposium on parallel and distributed processing*, Atlanta, GA, USA, April 2010, pp. 1–12.

Lea D. Dlmalloc. ftp://gee.cs.oswego.edu/pub/misc/malloc.c. Accessed Aug 2010.

LeVasseur J, Uhlig V, Stoess J and Götz S (2004) Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th symposium on operating systems design and implementation*, San Fransisco, CA, USA, December 2004, pp. 17–30.

Liedtke J (1995) On μ-kernel construction. In *Proceedings of the 15th symposium on operating system principles*, Copper Mountain, CO, USA, December 1995, pp. 237–250.

Liedtke J, Uhlig V, Elphinstone K, Jaeger T and Park Y (1999) How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service-neutrality. In *Proceedings of 7th workshop on hot topics in operating systems*, Rio Rico, AZ, USA, March 1999, pp. 191–196.

Mergen MF, Uhlig V, Krieger O and Xenidis J (2006) Virtualization for high-performance computing. *ACM Operating Systems Review* 40: 8–11.

Minnich R and McKie J (2009) Experiences porting the Plan 9 research operating system to the IBM Blue Gene supercomputers. *Computer Science – Research and Development* 23: 117–124.

Moreira J, Brutman M, Castanos J, Gooding T, Inglett T, Lieber D, McCarthy P, Mundy M, Parker J, Wallenfelt B, Giampapa M, Engelsiepen T and Haskin R (2006) Designing a highly-scalable operating system: The Blue Gene/L story. In *Proceedings of the 2006 international conference on supercomputing*, Tampa, FL, USA, November 2006, pp. 53–63.

Netperf Team. Netperf. http://www.netperf.org/netperf/. Accessed Aug 2010.

Pratt I, Fraser K, Hand S, Limpach C, Warfield A, Magenheimer D, Nakajima J and Malick A (2005) Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux symposium*, Ottawa, Canada, July 2005, pp. 65–78.

Steinberg J and Kauer B (2010) NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th ACM SIGOPS EuroSys conference*, Paris, France, April 2010, pp. 209–221.

Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FCM, Anderson AV, Bennett SM, Kägi A, Leung FH and Smith L (2005) Intel virtualization technology. *IEEE Computer* 38(5): 48–56.

Van Hensbergen E (2006) PROSE: partitioned reliable operating system environment. *ACM Operating Systems Review* 40(2): 12–15.

Author's biographies

Jan Stoess is a senior researcher at Karlsruhe Institute of Technology, where he pursues research in the areas of operating systems, cloud computing, high-performance computing, and big data. Jan is also co-founder of HStreaming, a real-time big data company. Jan holds a PhD in Computer Science from the University of Karlsruhe, Germany. As part of his graduate program he worked at IBM Watson Research and at VMWare R&D.

Udo Steinberg is a researcher and PhD student in TU Dresden's operating systems group. He holds a diploma in computer science from TU Dresden and has several years of experience designing micro-kernel-based systems with a focus on virtualization, security, and real-time technologies. Udo is the architect of the NOVA microhypervisor and has interned at the Intel Oregon Research Center (Hillsboro, OR) and IBM's Thomas J Watson Research Center (Yorktown Heights, NY), where he worked on x86 and BlueGene/P virtualization projects, respectively.

Volkmar Uhlig is interested and has worked in many areas of systems including operating systems, distributed systems, virtualization, and big data. He is a co-founder of HStreaming, a real-time big data company. Previously, he spent five years at IBM's Thomas J Watson Research Center working on, among others things, cloud computing utilizing IBM's Blue Gene supercomputer and a massive-scale low-latency stream-processing platform. Prior to IBM, he was the lead architect of the L4 micro-kernel, which has been commercially deployed in over a billion cell phones. Volkmar has published numerous papers, holds several patents, and actively participates in the research community. He holds a PhD in Computer Science from the University of Karlsruhe, Germany.

Jens Kehne is a graduate student at Karlsruhe Institute of Technology. His research interests are operating systems and high-performance computing. He is currently working on his diploma thesis focusing on a communication facility for high-performance cloud operating systems. Jens holds a master's degree from Georgia Institute of Technology.

Jonathan Appavoo is an assistant professor in the Department of Computer Science at Boston University. His research focuses on two areas. His first area of focus is on how the power of today's supercomputers can be brought into the mainstream of computing to enable breakthrough applications that exploit the thousands of processors, terabytes of RAM, and the unprecedented networking capabilities available on these classes of computers. His second area of focus is on exploiting brain-inspired architectures to break the barriers faced by future general-purpose multiprocessor systems. Specifically, building on observations during his PhD at the University of Toronto and work at IBM, he is exploring a new systems architecture for general-purpose brain-based computing. Jonathan holds a PhD from the University of Toronto.

Amos Waterland is a PhD student at Harvard. His research interests include large parallel computer systems and machine learning.