Popcorn: a replicated-kernel OS based on Linux

Antonio Barbalace
Virginia Tech
antoniob@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

David Katz
Virginia Tech
dgk8293@vt.edu

Abstract

In recent years, the number of CPUs per platform has continuously increased, affecting almost all segments of the computer market. Because of this trend, many researchers have investigated the problem of how to scale operating systems better on high core-count machines. While many projects have used Linux as a vehicle for this investigation, others have proposed new OS designs. Among them, the replicated-kernel OS model, specifically the multikernel, has gained traction. In this paper, we present Popcorn: a replicated-kernel OS based on Linux. Popcorn boots multiple Linux kernel instances on multicore hardware, one per core or group of cores. Kernels communicate to give to applications the illusion that they are running on top of a single OS. Applications can freely migrate between kernels, exploiting all the hardware resources available on the platform, as in SMP Linux.

1 Introduction

In recent years, the number of CPUs per platform has continuously grown, affecting almost all segments of the computer market. After it was no longer practical to increase the speed of a processor by increasing its clock frequency, chip vendors shifted to exploiting parallelism in order to maintain the rising performance that consumers had come to expect. Nowadays, multiple chips, each containing multiple cores, are being assembled into single systems. All cores, across the different chips, share the same physical memory by means of cache coherency protocols. Although researchers were skeptical that cache coherence would scale [6] the multi core market continues to grow. Multi core processors are ubiquitous, they can be found in embedded devices, like tablets, set top boxes, and mobile devices (e.g., Exynos Octa-Core [3]), in home/office computers (e.g., AMD Fusion, Intel Sandy Bridge), in high-end servers (e.g., AMD Opteron [13], Intel Xeon [1]), and in HPC machines (e.g., SGI Altix [2]). These types of multiprocessor systems, formerly available only as high cost products for the HPC market, are today more affordable and are present in the consumer market.
Because of this trend, many researchers have investigated the problem of how to better scale operating systems on high core count machines. While some projects have used Linux as a vehicle for this investigation [6, 7], others have proposed new operating system (OS) designs [5]. Among them, the replicated-kernel OS model has gained traction.

Linux has been extended by its large community of developers to run on multiprocessor shared memory machines. Since kernel version 2.6, preemption patches, ticket spinlocks, read/write locks, and read-copy-update (RCU) have all been added. Several new techniques have also been added to improve data locality, including per_cpu infrastructure, the NUMA-aware memory allocator, and support for scheduling domains. B. Wickizer *et al.* [6] conclude that vanilla Linux, on a large-core-count machine, can be made to scale for different applications if the applications are carefully written. In [7] and [12] the authors show that scalable data structures, specifically scalable locks, like MCS, and RCU balanced tree, help Linux scale better when executing select applications.

Although recent research has demonstrated Linux's scalability on multicore systems to some extent, and Linux is already running on high core count machines (e.g., SGI Altix [2]) and accelerators (e.g., Intel Xeon-Phi [15]), it is important to understand whether Linux can be used as the basic block of a replicated-kernel OS. Understanding the advantages of this OS architecture on Linux – not just from a scalability standpoint – is important to better exploit the increasingly parallel hardware that is emerging. If future processors do not provide high-performance cache coherence, Linux's shared memory intensive design may become a significant performance bottleneck [6].

The replicated-kernel OS approach, advanced in operating systems including Hive, Barrelfish, FOS, and others, is a promising way to take advantage of emerging high-core count architectures. A. Baumann *et al.* [5] with Barrelfish introduced the term multikernel OS and showed appealing scalability results demonstrating that their design scales as well, if not better, than SMP Linux on selected applications up to 32 cores. A multikernel OS is an operating system that is made up of different (micro-) kernel instances, each of which runs on a single core of a multi core device. Kernels communicate in order to cooperatively maintain partially replicated OS state. Each kernel runs directly on bare hardware, no (hardware or software) virtualization layer is employed. Because Popcorn does not adhere to such definition we use the term replicated-kernel OS to identify a broader category of multikernels, including Popcorn.

Popcorn In this paper, we present Popcorn: a replicated-kernel OS based on Linux. Popcorn boots multiple Linux kernel instances on multicore hardware, one per core or group of cores, with kernel-private memory and hardware devices. The kernel instances directly communicate, kernel-to-kernel, in order to maintain a common OS state that is partially replicated over every individual kernel instance. Hardware resources (i.e., disks, network interface cards) are fully shared amongst the kernels. Kernel instances coordinate to maintain the abstraction of a single operating system (single system image), enabling traditional applications to run transparently across kernels. Inter-kernel process and thread migration are introduced to allow application threads to transparently execute across the kernels that together form the OS. Considering that vanilla Linux scales well on a bounded number of cores, we do not put any restrictions on how many cores the same kernel image will run on.

Contributions Our primary contribution is an open-source replicated-kernel OS using Linux as its basic building block, as well as its early evaluation on a set of benchmarks. To the best of our knowledge this is the first attempt in applying this design to Linux.
Multiple Popcorn kernels, along with the applications that they hosts, can simultaneously populate a multi core machine. To facilitate this, we augmented the Linux kernel with the ability to run within a restricted subset of available hardware resources (e.g. memory). We then strategically partition those resources, ensuring that partitions do not overlap, and dedicate them to single kernel instances.

To create the illusion of a single operating system on top of multiple independent kernel instances we introduced an inter-kernel communication layer, on top of which we developed mechanisms to create a single system image (e.g. single filesystem namespace) and inter-kernel task migration (i.e. task and address space migration and address space consistency). TTY and a virtual network switch was also developed to allow for communication between kernels. Our contribution also includes a set of user-space libraries and tools to support Popcorn. A modified version of *kexec* was introduced to boot the environment; the *util* toolchain was built to create replicated-kernel OS configurations. MPI-Popcorn and the *cthread/pomp* library were introduced to support MPI and OpenMP applications on Popcorn, respectively.
Here we describe Popcorn's architecture and implementation details, in particular, the modifications that we introduced into the Linux source code to implement the features required by a replicated-kernel OS design. We also present initial results obtained on our prototype, which was developed on x86 64bit multicore hardware. Popcorn has been evaluated through the use of cpu/memory intensive, as well as I/O intensive workloads. Results are compared to results from the same workloads collected on SMP Linux and KVM.

Document Organization We first present the existing work on the topic in Section 2. We introduce our design choices and architecture in Section 3, and we cover the implementation details of our prototype in Section 4. We present the experimental environment in Section 5 and discuss the results we obtained by running selected benchmarks on Popcorn against mainly vanilla Linux (called *SMP Linux* hereafter) in Section 6. Finally, we conclude in Section 7.

2 Related Work

The body of work related to our approach includes contributions in operating systems, distributed and cluster systems, and Linux design and performance measurements. We leveraged ideas and experience from these different efforts in order to build on their findings and to address their limitations where possible.

Non-Linux based Several decades ago, Hurricane [22] and Hive [11] ('92 and '95) introduced the idea of a replicated-kernel OS by means of clusters or cells (in Hive) of CPUs sharing the same kernel image. This is different from common SMP operating systems, where a single kernel image is shared by all the CPUs. While these approaches were built to work on research hardware, the multikernel model was recently revisited by Barrelfish [5] on modern multicore commodity hardware, with each core loading a single kernel image. A similar approach was taken by FOS [23] addressing emerging high core-count architectures where computational units do not share memory. All these approaches use message passing for inter-kernel communication. These message passing mechanisms are implemented using shared memory programming paradigm. Popcorn follows the same approach but differs in the way the communication mechanism is implemented and in which data structures are kept consistent amongst kernels. Hurricane, Barrelfish and FOS are microkernel-based, but Hive was developed as a modification of the IRIX variant of Unix, similarly to how our work is based on Linux.

Linux-based approaches To the best of our knowledge there is no previous effort that uses Linux as the kernel block in a replicated-kernel OS. However, there are notable efforts in this direction that intersect with the cluster computing domain, including ccCluster (L. McVoy), K. Yaghmour's work [24] on ADEOS, and Kerrighed [18]. Kerrighed is a cluster operating system based on Linux, it introduced the notion of a kernel-level single system image. Popcorn implements a single system image on top of different kernels as well, but the consistency mechanism and the software objects that are kept consistent are fundamentally different. ccCluster and ADEOS aim to run different operating systems on the same hardware on top of a nano-kernel, cluster computing software was used to offer a single system image. Popcorn kernels run on bare hardware.

We know of three separate efforts that have implemented software partitioning of the hardware (i.e. multiple kernel running on the same hardware without virtualization) in Linux before: Twin Linux [16], Linux Mint [19] and SHIMOS [21]. Different from Popcorn, the source code of these solutions are not available (even upon request). Although they describe their (different) approaches, they do not present significant performance numbers, and they do not explore the functionality of their solution over 4 CPUs (on the x86 architecture). Twin Linux was deployed on a dual-core processor. The authors modified GRUB in order to boot two instances of Linux in parallel using different images and hardware resource partitions on different cores. To allow the kernels to communicate with each other, they provide a shared memory area between the kernels. Linux Mint, despite similarities to Twin Linux, lacks an inter-kernel communication facilities. The bootup of different kernel instances in Linux Mint is handled sequentially by the bootstrap processor, something we borrow in our implementation of Popcorn. Popcorn and SHIMOS boot different kernel instances sequentially but any kernel can boot any other kernel. SHIMOS implements an inter-kernel communication mechanism in order to share hardware resources between different Linux instances. Despite the fact that the same functionality is implemented in Popcorn, SHIMOS was designed as a lightweight alternative to virtualization. Therefore it does not implement all of the other features that characterize a multikernel OS.

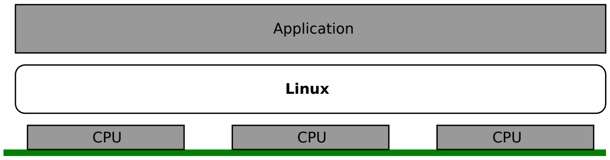

Figure 1: SMP Linux software architecture on multi core hardware. A single kernel instance controls all hardware resources and manages all applications.

3 Popcorn Architecture

The replicated-kernel OS model, mainly suitable for multi-core hardware, implies a different software architecture than the one adopted by SMP Linux. The SMP Linux software stack is depicted in Figure 1. A single kernel instance controls all hardware resources. Applications run in the user space environment that the kernel creates. Popcorn's software stack is shown in Figure 2. Each kernel instance controls a different private subset of the hardware. New software layers have been introduced to allow an application to exploit resources across kernel boundaries. These new software layers are addressed in this section.

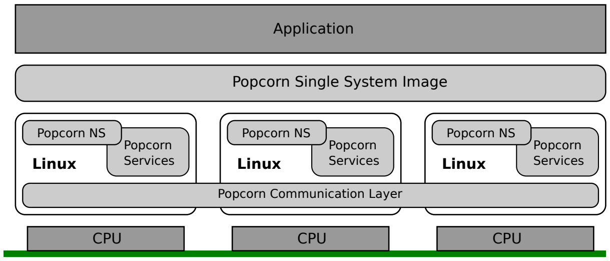

Figure 2: Popcorn Linux software architecture. Each core or group of cores loads a kernel instance. Instances communicate to maintain a single system image.

3.1 Software Partitioning of the Hardware

The idea of running different kernels on the same machine is nowadays associated with virtualization technologies. The replicated-kernel OS model does not imply a virtualization layer. In a virtualized environment different *guest* kernels coexist on top of a hypervisor. Several virtualization solutions (e.g., Xen, KVM) rely on the presence of one of the kernels, the *host*, for services, drawing on a hierarchical relationship between them. There is no hypervisor in our approach. Instead, all kernel instances are peers that reside within different resource partitions of the hardware. Thus, services can run (virtually) anywhere. Without the hypervisor enforcing hardware resource partitioning and managing hardware resource sharing among kernels, the Linux kernel itself should be able to operate with any subset of hardware resources available on the machine. Therefore we added software partitioning of the hardware as first class functionality in Popcorn Linux.

CPUs Within SMP Linux, a single kernel instance runs across all CPUs. After it is started on one CPU, as a part of the initialization process, it sequentially starts all the other CPUs that are present in the machine (Figure 1 and Figure 3.a). Within Popcorn, multiple kernel instances run on a single machine (Figure 2). After an initial kernel, named primary, has been booted up on a subset of CPUs (Figure 3.b), other kernels, the secondaries, can be booted on the remaining CPUs (Figure 3.c and 3.d). Like in SMP Linux each Popcorn kernel instance boots up on single CPU (the bootup) and eventually brings up a group of application CPUs. In Figure 3.b *Processor 0 Core 0* is the bootup CPU and starts *Processor 0 Core 1*. We support arbitrary partitioning and clustering of CPUs, mapping some number of CPUs to a given kernel instance. Partitioning refers to assigning one kernel instance per processor core (Figure 3.c and 3.d); clustering refers to configurations where a single kernel image runs on multiple processor cores (Figure 3.b). CPUs that are working together in a kernel instance do so in SMP fashion.

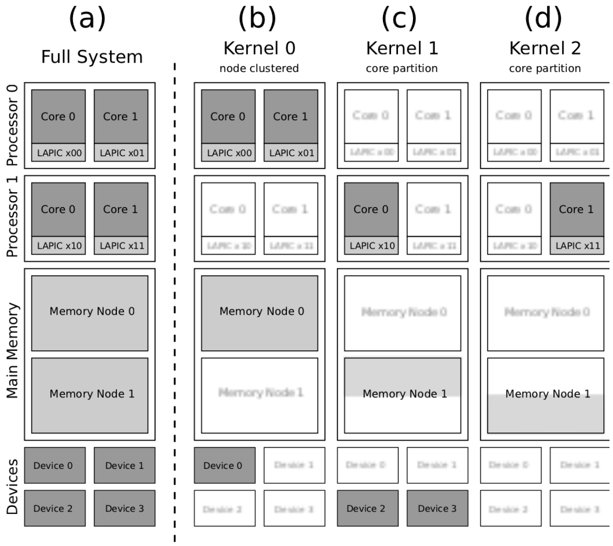

Figure 3: Partitioning and clustering of hardware resources. A kernel instance can be assigned to different a single core or a group of cores. Memory is partitioned on node boundaries (when possible).

Memory and Devices Linux is an SMP OS, as such, when loaded it assumes that all hardware resources must be discovered and loaded; i.e. all resources belong to one kernel instance. In Popcorn, different kernels should coexist, so we start each kernel with a different subset of hardware resources; enforced partitioning is respected by each kernel. Popcorn partitioning includes CPUs (as described above), physical memory and devices. Every kernel owns a private, non overlapping, chunk of memory, and each device is assigned at startup to one only kernel.

In Figure 3 depicts an example of partitioning of 4 devices to 3 kernels. Figure 3.b shows that *Device 0* is owned by *Kernel 0*; Figure 3.c shows that *Device 2* and *Device 3* are own by *Kernel 1*.

On recent multi-core architectures, each group of cores has a certain amount of physical memory that is directly connected to it (or closely bounded). Thus, accessing the same memory area from different processors

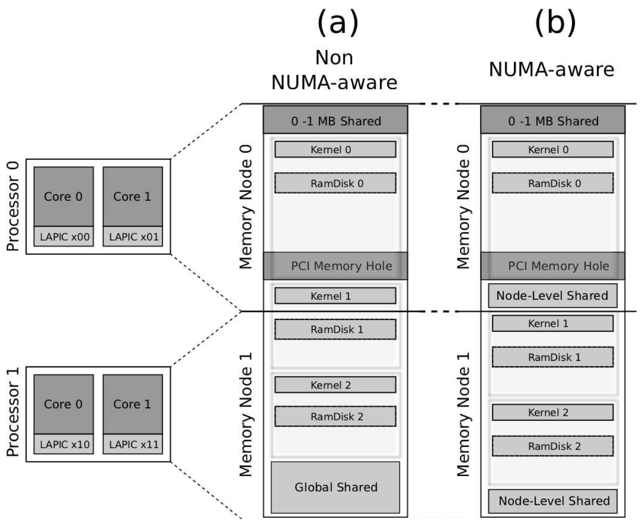

Figure 4: Non-NUMA-aware (a) and NUMA-aware (b) partitioning/clustering of memory on the x86 architecture. The PCI memory hole and the sharing of the first 1 MB of RAM are enforced by the architecture.

incurs different amounts of latencies (NUMA). Physical RAM is usually contiguous but it can contain memory holes, for example to map hardware devices (e.g. PCI hole in x86). Moreover, depending on host architecture, parts of the address space must be shared between all kernels (e.g. in x86 the first 1MB of memory should be shared because of the boot sequence). The partitioning of physical memory must consider all of these architecture-specific details. We developed a variety of memory partitioning policies that apply to CPU partitions or clusters. Figure 4 shows two possible memory policies: non-NUMA-aware (Figure 4.a), and NUMA-aware (Figure 4.b). The latter gives greatest weight to deciding how to allocate private memory windows to the system topology, with the aim of reducing memory access latencies.

3.2 Inter-Kernel Communication

A replicated-kernel OS, strives to provide a single execution environment amongst kernels. In order to accomplish this, kernels must be able to communicate. In Popcorn, communication is based on message passing.

To let the kernels communicate we introduced a low-level message passing layer, deployed over shared memory. Many message passing techniques have been introduced in the literature [9, 10, 5]. We opted for a communication mechanism with private receive-only buffers. Such buffers are allocated in the receiver kernel memory. The layer provides priority-based, synchronous and asynchronous messaging between kernels.

It also provides multicast capabilities to allow for one to many communications. This capability is useful in implementing many distributed algorithms, including distributed locks, voting mechanisms, and commit protocols.

3.3 Single System Image

Applications running on SMP Linux expect a single system image regardless of the CPU they are running on. That is all the CPUs, peripheral devices, memory, can be used by all applications concurrently; furthermore processes communicate and synchronize between them. In Popcorn, the messaging layer is used to coordinate groups of kernels to create a single working environment. Similar to the pioneering work in Plan9 [20], Popcorn's single system image includes: single filesystem namespace (with devices and proc), single process identification (PID), inter-process communication (IPC), and CPU namespaces.

Relative to physical memory and available CPUs, hardware peripherals are comparatively limited in number. After being initialized by a kernel they cannot be accessed in parallel by different kernels; for the same reason concurrent access to devices in SMP Linux is synchronized through the use of spinlocks. Popcorn makes use of a master/worker model for drivers to make devices concurrently accessible through any kernel via the single system image. The master kernel owns the device, and the worker kernels interact with the device through message exchange with the master kernel.
A specific example of such a device is the I/O APIC, the programmable interrupt controller on the x86 architecture. The I/O APIC driver is loaded exclusively on the primary kernel, that becomes the master, and any other kernels with a device in their resource partition that requires interrupts registration exchange messages with that kernel in order to operate with the I/O APIC.

3.4 Load Sharing

In a multikernel OS, as in a SMP OS, an applications' threads can run on any of the available CPUs on the hardware. Popcorn was extended to allow user-space tasks to arbitrarily migrate between kernels. Most of the process related data structures are replicated, including the virtual address space. The virtual address space for a process with threads executing on different kernels is maintained consistent over the lifetime of the process.

4 x86 Implementation

Linux was not designed to be a replicated-kernel OS. In order to extend its design, large parts of Linux had to be reengineered. In this section we cover implementation details that characterize our x86 64bit prototype. We deployed Popcorn starting from vanilla Linux version 3.2.14. The current Popcorn prototype adds a total of $\sim 31k$ lines to the Linux kernel source. The user-space tools are comprised of $\sim 20k$ lines of code, of which $\sim 4k$ lines were added to *kexec*.

4.1 Resource Partitioning

Loading Popcorn Linux requires the establishment of a hardware resource partitioning scheme prior to booting the OS. Furthermore, after the primary kernel has been booted up, all of the remaining kernels can be started. Once this procedure has been followed, the system is ready to be used, and the user can switch to the Popcorn namespace and execute applications on the replicated-kernel OS. Figure 5.a illustrates the steps involved in bringing up the Popcorn system - from OS compilation to application execution. If the *Hotplug* subsystem is available, the steps in Figure 5.b can be followed instead.

In support of partitioning, We have built a chain of applications that gather information on the machine on which Popcorn will run. That information is then used to create the configuration files needed to launch the replicated-kernel environment. This tool chain must be run on SMP Linux, on the target machine, since it exploits resource enumeration provided by the Linux kernel's NUMA subsystem. A vast set of resource partitioning rules can be used to generate the configuration, including and not limited to one-kernel per core, one-kernel per NUMA node, same amount of physical memory per kernel, memory aligned on NUMA boundary. The configuration parameters that are generated are mostly in the form of kernel command-line arguments.

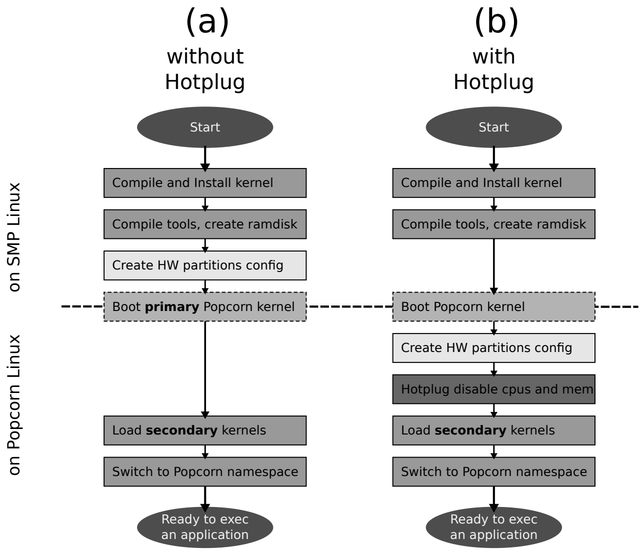
(a)
without
Hotplug
(b)
with
Hotplug
on SMP Linux
on Popcorn Linux
Start
Compile and Install kernel
Compile tools, create ramdisk
Create HW partitions config
Boot **primary** Popcorn kernel
Load **secondary** kernels
Switch to Popcorn namespace
Ready to exec an application
Start
Compile and Install kernel
Compile tools, create ramdisk
Boot Popcorn kernel
Create HW partitions config
Hotplug disable cpus and mem
Load **secondary** kernels
Switch to Popcorn namespace
Ready to exec an application

Figure 5: Two different ways of booting Popcorn Linux. Without and with hotplug (CPU and memory) support.

4.2 Booting Multiple Kernels

The *kexec* software is normally used to reboot a machine into a new kernel from an already-running kernel. We modified both the *kexec* application itself and the backend code in the Linux kernel to load new kernel instances (secondary kernels) that run in parallel with the current one, but on a different partition of hardware resources. Kernels are booted in sequence.

As part of the *kexec* project, the Linux kernel was made relocatable, and can potentially be loaded and executed from anywhere within the physical address space [14]. However, it was necessary to rewrite the kernel bootup code in *head_64.S* to boot kernels at any location throughout the entire physical address space. This modification required an addition to the early pagetable creation code on x86 64bit. In order to boot secondary kernels, we modified the trampoline, that is the low level code used to boot application processors in SMP Linux. We added *trampoline_64_bsp.S* whose memory space gets reserved in low memory at boot time. The same memory range is shared by all kernels. Because the trampolines are used by all CPUs, kernels should boot in sequences. The structure `boot_param` has been also modified in order to support a ramdisk sit everywhere in the physical address space.

The boot sequence of a secondary kernel is triggered by a syscall to the *kexec* subsytem. A kernel image is specified in a syscall argument. Rather than reusing the same

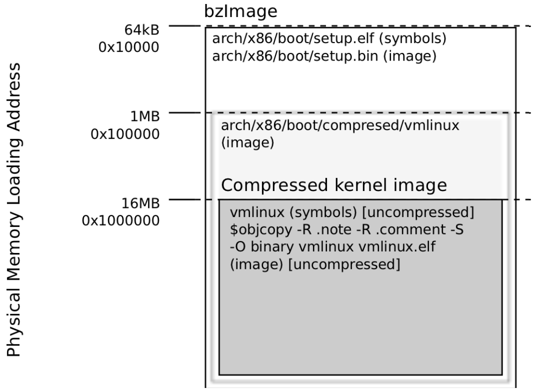
bzImage
64kB
0x10000
arch/x86/boot/setup.elf (symbols)
arch/x86/boot/setup.bin (image)
1MB
0x100000
arch/x86/boot/compresed/vmlinux
(image)
Compressed kernel image
16MB
0x1000000
vmlinux (symbols) [uncompressed]
$objcopy -R .note -R .comment -S
-O binary vmlinux vmlinux.elf
(image) [uncompressed]
Physical Memory Loading Address

Figure 6: Linux's bzImage objects loading addresses. bzImage is a compressesed kernel image. It self-extracts the *vmlinux* binary exectuable at 16MB.

bzImage used by the primary kernel to bring up the machine, a decompressed and stripped kernel binary is used to boot the secondary kernels. Figure 6 shows the organization of the `bzImage` format. We extract and boot the "Compressed kernel image", i.e. a stripped version of `vmlinux`, to keep the size bounded and speed up the boot process.

The modified *kexec* copies the kernel binary and the boot ramdisk to the selected physical locations. It then initializes the secondary kernel's `boot_params` with the appropriate kernel arguments and ramdisk location/size. At this point *kexec* sets the remote CPU's initial instruction pointer to point to the Popcorn trampoline, and sends an inter-processor interrupt (IPI) to the CPU to wake up. After the remote CPU executes its trampoline, which loads Popcorn' 64bit identity-mapped pagetable for the appropriate region, the CPU is able to start a kernel located anywhere in the physical address space.

4.3 Partitioning CPUs

In SMP Linux, each CPU receives an incremental logical identifier, starting from 0, during boot. This logical id can be acquired by kernel code with a call to `smp_processor_id()`. This identifier is separate from the local APIC identifier adopted in the x86 architecture to distinguish processor cores. In a multicore machine Popcorn tries to keep the same CPU enumeration as is used in SMP Linux; for example CPU identifiers of *Kernel 0* and *Kernel 1* in Figure 3.b and 3.c will be $0, 1$ and 2 respectively (not $0, 1$ and 0). This was achieved by relying on the APIC ids and the ACPI tables passed in by the BIOS.

We furthermore modified the Linux source to upgrade the legacy subsystems initialization, based on a check on the kernel id, to a more generic mechanism. Many kernel subsystems were initialized only if `smp_processor_id()` was 0. In Popcorn, a new function has been added that returns true if the current CPU is booting up the kernel. Another function, `is_lapic_bsp()`, reveals whether the kernel is the primary.

In order to select which CPUs are assigned to a kernel we introduce the kernel command line argument `present_mask`, which works similarly to `possible_mask` added by the *Hotplug* subsystem. Booting a kernel on a cluster of cores can be done by choosing any combination of core ids. It is not necessary that they are contiguous.

4.4 Partitioning Memory and Devices

A resource-masking feature was implemented to let primary and secondary kernels boot with the same code, on the same hardware. Linux comes with a set of features to include and exclude memory from the memory map provided by the BIOS. We exploited the `memmap` family of kernel command line arguments to accomplish memory partitioning.

In Popcorn, we restrict each kernel to initialize only the devices in its resource partition. When SMP Linux boots, it automatically discovers most of the hardware devices present on the system. This process is possible by means of BIOS enumeration services and dynamic discovery. Dynamic discovery is implemented in several ways, e.g. writing and then reading on memory locations or I/O ports, writing at an address and then waiting for an interrupt. This feature is dangerous if executed by kernel instances other than the kernel that contains a device to be discovered in its resource partition. Currently only the primary kernel has dynamic discovery enabled.

BIOS enumeration services provide lists of most of the devices present in the machine (in x86 ACPI). Each kernel in our system has access to these lists in order to dynamically move hardware resources between running kernels. If the static resource partition of a kernel instance does not include a hardware resource, this resource must not be initialized by the kernel. For example, in the PCI subsystem, we added a blacklisting capability to prevent the PCI driver from initializing a device if it was blacklisted. In our implementation, the blacklist must be provided as a kernel command line argument (`pci_dev_flags`).

4.5 Inter-Kernel Message Passing

A kernel-level message passing layer was implemented with shared memory and inter processor interrupt (UPI) signaling to communicate between kernel instances. The slot-based messaging layer uses cache-aligned private buffers located in the receivers memory. The buffering scheme is multiple writer single reader; concurrency between writers is handled by means of a ticketing mechanism. After a message has been written into a buffer, the sender notifies the receiver with an IPI. In order to mitigate the inter processor traffic due to IPIs, our layer adopts a hybrid of polling and IPI for notification. While the receiver dispatches messages to tasks after receiving a single IPI and message, senders can queue further messages without triggering IPIs. Once all messages have been removed from the receive buffer, IPI delivery is reinstated.

A multicast messaging service is also implemented. In order for a task to send a multicast message it first should open a multicast group. Every message sent to the group is received by the groups subscribers. Multicast groups are opened and closed at runtime. When a message is sent to a group, it is copied to a memory location accessible by all subscribers and an IPI is sent to each of them iteratively. Because in a replicated-kernel OS different kernels coexist on the same hardware, we disable IPI broadcasting (using the kernel command line argument `no_ipi_broadcast`). IPI broadcasting will add additional overhead if used for multicast notification. Nonetheless, the hardware we are using does not support IPI multicast (x2 APIC).

The message passing layer is loaded with `subsys_initcall()`. When loaded on the primary kernel, it creates an array of buffer's physical addresses (`rkvirt`, refer to Figure 7). These arrays are populated by each of the kernels joining the replicated-kernel OS with their respective receiver buffer addresses during their boot processes. The address of this special array is passed via `boot_param` struct to each kernel. Every time a new kernel joins the replicated-kernel OS it adds its receiver buffer address to `rkvirt` first, and then communicates its presence to all the other registered kernels by message.

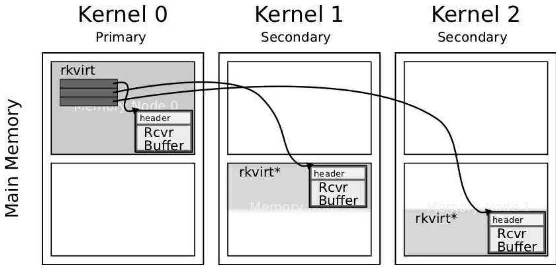

Figure 7: Receiver buffers are allocated in the kernel's private memory. In gray is the private memory of each kernel. `rkvirt`, the array holding the addresses of the receiving buffers, is allocated on the primary kernel.

4.6 Namespaces

Namespaces were introduced into SMP Linux to create sub environments, as a lightweight virtualization alternative implemented at the OS level. Kerrighed [18] uses namespaces (containers) to migrate applications in a cluster by reproducing the same contained environment on different kernels. Popcorn uses namespaces to provide a single environment shared between kernels. Each kernel's namespace is kept consistent with the others' through the messaging layer.

Linux 3.2.14, on top of which we developed Popcorn, supports uts, mount, IPC, PID and network namespaces, though the API is incomplete. Hence code has been back ported from Linux kernel 3.8. Popcorn was extended to include a *CPU namespace* (see below). Mechanisms were also added to create namespaces that are shared between kernels. Currently the mount namespace relies on NFS. The network namespace is used to create a single IP overlay.

After kernels connect via the messaging layer, static Popcorn namespaces are created in each kernel. A global nsproxy structure is then made to point to Popcorn's uts, mount, IPC, PID, network and CPU namespace objects. Namespaces on each kernel are updated whenever a new kernel joins the replicated-kernel OS. Because Popcorn namespaces get created by an asynchronous event rather than the creation of a task, instead of using the common */proc/PID/ns* interface we added */proc/popcorn*. Tasks join Popcorn by associating to its namespaces through the use of the setns syscall, which has been updated to work with statically created namespaces. When a task is migrated to another kernel it starts executing only after being (automatically) associated to Popcorn namespaces.

CPU Namespace An application that joins Popcorn's CPU namespace can migrate to any kernel that makes up the replicated-kernel OS, i.e., by mean of sched_setaffinity(). Outside Popcorn's CPU namespace, only the CPUs on which the current kernel is loaded are available to applications. If Linux's namespaces provides a subset of the resources (and for CPUs Linux includes cgroups or domains), Popcorn's namespaces show a superset.

We added the kernel command line parameter cpu_offset. This is the offset of the current cpumask_t in the Popcorn CPU namespace. We added the field cpuns of type struct cpu_namespace * to the struct nsproxy. We augment task_struct with a new list_head struct to hold a variable length cpu bitmap. Amongst the other, the functions sched_getaffintiy and sched_setaffinity have been updated to work based on the current CPU namespace. Finally, a task in a different CPU namespace will read different contents from */proc/cpuinfo*. In the Popcorn namespace it will see all the CPUs available in all joining kernels.

4.7 Devices

Following the peer kernel paradigm, implied by the replicated-kernel OS design, we adopted inter-kernel coordination in order to access remotely owned hardware resources. Devices rely on namespaces for enumeration purposes, and message passing for access and coordination. Access to a device can be proxied by a kernel (e.g. the I/O APIC example from Section 3.3) otherwise ownership of a device can be passed to another kernel (in the case where exclusive ownership is required, i.e. CD-ROM burning application). On kernels in which a device is not loaded, a dummy device driver monitors application interaction with that device. Based on the device type, application's requests are either forwarded to the owner kernel (proxied access), or locally staged waiting for local device driver re-initialization (ownership passing).

The following inter-kernel communication devices were implemented outside the messaging dependent mechanism described above for debugging and performance reasons.

Virtual TTY For low-bandwidth applications (e.g. launching processes or debugging) a virtual serial line device, controlled by a TTY driver, is provided between any kernel pair. Each kernel contains as many virtual TTY device nodes (*/dev/vtyX*) as there are kernels in Popcorn (*X* is the smallest CPU id of the kernel to connect to). Each kernel opens a login console on */dev/vty0* during initialization. A shared memory region is divided between all of the device nodes in a bidimensional matrix. Each cell of the matrix holds a ring buffer and the corresponding state variables. The reading mechanism is driven by a timer that periodically moves data from the shared buffer to the flip buffer of the destination kernel.

Virtual Network Switch As in virtual machine environments, we provide virtual networking between kernel instances. The kernel instance that owns the network card acts as the gateway and routes traffic to all the other clients. In this setup each kernel instance has an associated IP address, switching is automatically handled at the driver level. A network overlay, constructed using the network namespace mechanism, provides a single IP amongst all kernels. We developed a kernel level network driver that is based on the Linux TUN/TAP driver but uses IPI notification and fast shared-memory ring buffers for communication. Our implementation uses the inter-kernel messaging layer for coordination and check-in (initialization).

4.8 Task Migration

To migrate tasks, i.e. processes or threads, between kernel instances, a client/server model was adopted. On each kernel, a service is listening on the messaging layer for incoming task migrations. The kernel from which a task would like to out-migrate initiates the communication.

An inter-kernel task migration comprises of three main steps. First, the task which is to be migrated is stopped. Secondly, the whole task state is transferred to the server, where a dummy task, that acts as the migrated task, is created on the remote kernel. Thirdly, all the transferred information about the migrated task are imported into the dummy task, and the migrated task is ready to resume execution.

The task that was stopped on the sending kernel remains behind, inactive, as a *shadow task*. A shadow task is useful for anchoring resources, such as memory and file descriptors, preventing reuse of those resources by the local kernel. Shadow tasks also serve the purpose of speeding up back migrations. When a task is migrated back to a kernel that it has already visited, its current state is installed in the shadow task, and the shadow task is reactivated.

Task State The state of a task is comprised of register contents, its address space, file descriptors, signals, IPCs, users and groups credentials. Popcorn currently supports migrating `struct pt_regs` and `union thread_xstate`, as CPU's registers on the x86 architecture. The address space must also be migrated to support the continued execution of migrated tasks. An address space is comprised of virtual memory area information `struct vm_area_struct`, and the map of physical pages to those virtual memory areas. The latter can obtained by walking the page tables (using `walk_page_range`). Address space mappings are migrated on-demand, in keeping with Linux custom. Mappings are migrated only in response to fault events. This ensures that as a task migrates, overhead associated with carrying out mapping migrations is minimized by migrating only mappings that the task needs. When an application needs a mapping, that mapping is retrieved from remote kernels, and replicated locally. Page level granularity is supported. If no mapping exists remotely, one is created using the normal Linux fault handling routine. This is how tasks come to rely on memory owned by multiple kernels. As a task migrates and executes, it's memory is increasingly composed of locally owned memory pages and remote owned memory pages (the latter do not have an associated `struct page`, therefore are not normal pages). Remote pages are guaranteed to remain available due to the presence of shadow tasks. A configurable amount of address space prefetch was also implemented, and found to have positive performance effect in some situations. Prefetch operations are piggy-backed on mapping retrieval operations to reduce messaging overhead.

State Consistency No OS-level resources are shared between tasks in the same thread group which happen to be running on different kernels. Instead, those resources are replicated and kept consistent through protocols that are tailored to satisfy the requirements of each replicated component.

Inter-kernel thread migration causes partial copies of the same address space to live on multiple kernels. To make multi-threaded applications work correctly on top of these different copies, the copies must never contain conflicting information. Protocols were developed to ensure consistency as memory regions are created, destroyed, and modified, e.g. mmap, mprotect, munmap, etc.

File descriptors, signals, IPCs and credentials are also replicated objects and their state must also be kept consistent through the use of tailored consistency protocols.

5 Evaluation

The purpose of this evaluation is to investigate the behavior of Popcorn when used as 1) a tool for software partitioning of the hardware, and 2) a replicated-kernel OS. A comparison of these two usage modes will highlight the overheads due to the introduced software mechanism for communication, SSI and load sharing between kernels. We compared Popcorn, as a tool for software partitioning of the hardware, to a similarly configured virtualized environment based on KVM, and to SMP Linux. Popcorn as a replicated-kernel OS is compared to SMP Linux.

Hardware We tested Popcorn on a Supermicro H8QG6 equipped with four AMD Opteron 6164HE processors at 1.7GHz, and 64GB of RAM. Each processor socket has 2 physical processors (nodes) on the same die, each physical processor has 6 cores. The L1 and L2 caches are private per core, and 6 MB shared L3 cache exist per processor. All cores are interconnected cross-die and in-die, forming a quasi-fully-connected cube topology [13]. RAM is equally allocated in the machine; each of the 8 nodes has direct access to 8GB.

Software Popcorn Linux is built on the Linux 3.2.14 kernel; SMP Linux results are based on the same vanilla kernel version. The machine ran Ubuntu 10.04 Linux distribution, the ramdisk for the secondary kernels are based on Slackware 13.37.

In a first set of experiments we used the most recent version of KVM/Nahanni available from the project website [17]. We adopted *libvirt* (version 0.10.2) for managing the VMs. Since *libvirt* does not currently support *ivshmem* devices (Nahanni), we modified the *libvirt* source code. To run the MPI experiments, KVM/Nahanni includes a modified version of MPICH2/Nemesis called MPI-Nahanni (although not publicly available). The modified version exploits Nahanni shared memory windows for message passing. The code we received from the authors required some fixes to work properly. Based on MPI-Nahanni we implemented an MPI-Popcorn version of MPICH2/Nemesis.

The OpenMP experiments do not use *glibc/pthread* nor *glibc/gomp*. Instead a reduced POSIX threading library, without futexes, is adopted, called *cthread*. Furthermore *gomp* is replaced by a modified version of the custom OpenMP library derived from the Barrelfish project, we called it *pomp* as Popcorn OpenMP.

In all experiments we setup KVM/Nahanni with one virtual machine per core. Similarly, we configure Popcorn with one kernel per core. Linux runs with all available cores on the machine active but not when running the OpenMP experiments. In this case we set the number of active cores equal to the number of threads.

5.1 CPU/Memory Bound

To evaluate Popcorn on CPU/memory bounded workloads, we used NASA's NAS Parallel Benchmark (NPB) suite [4]. In this paper we present results from Class A versions of Integer Sort (IS), Conjugate Gradient (CG), and Fourier Transform (FT) algorithms. We chose Class A, a small data size, in order to better highlight operating system overheads. NPB is available for OpenMP (OMP) and MPI. The OMP version is designed for shared memory machines, while the MPI version is more suitable for clusters.

Because they use two different programming paradigms, OMP and MPI are not directly comparable. Because of that, we use both versions to quantify the overhead, compared to Linux, of the software partitioning of the hardware functionality, and the full software stack required by the replicated-kernel OS.

MPI A setup in which multiple kernel instances coexist on the same hardware resembles a virtualization environment. Therefore we decided to compare Popcorn, not only with SMP Linux but also with KVM/Nahanni [17] (that resemble the Disco/Cellular Disco replicated-kernel OS solution [8]).

Nahanni allows several KVM virtual machines, each running Linux, to communicate through a shared memory window. MPI applications are used in this test because despite there is shared memory, an OpenMP application can not run across multiple virtual machines. Because this test focuses on compute/memory workloads, we used MPI-Nahanni, which does use network communication, only for coordination, i.e. there is no I/O involved after the application starts. For this test Popcorn is not exploiting it's messaging layer, SSI, or load sharing functionality. MPI-Popcorn relies only on the presence of the standard */dev/mem*, although the virtual network switch is used to start the MPI application.

OpenMP As a replicated-kernel OS, Popcorn is able to transparently run a multithreaded application across multiple kernels. Therefore we compare the performance of the aforementioned NPB applications while running on SMP Linux and Popcorn. OMP NPB applications were compiled with *gcc* once with *cthread*, and *pomp*, and then run on both OSes. This experiment highlights the overhead due to all of Popcorn's software layers.

5.2 I/O Bound

This test investigates the response time of a web server running on our prototype in a secondary kernel. Because our inter-kernel networking runs below the SSI layer, this test stresses the software partitioning of the hardware functionality of Popcorn.

We run the event-based *nginx* web server along with ApacheBench, a page request generator from the Apache project. From a different machine in the same network, we generate http requests to SMP Linux, to secondary kernels in Popcorn, and to guest kernels on KVM. This configuration is shown in Figure 8. Although higher-performance network sharing infrastructures exist for KVM, we use the built-in bridged networking that is used in many research and industrial setups.

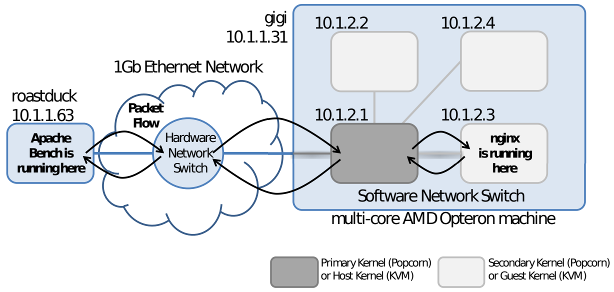

Figure 8: The network configuration used to generate the results in Figure 15.

6 Results

6.1 CPU/Memory workloads

MPI Figures 9, 10 and 11 show how Popcorn and its competitors perform on integer sort, the Fourier transform, and the conjugate gradient benchmarks, respectively, within MPI. Graphs are in log_2 scale. For each data point, we ran 20 iterations, and provide average and standard deviation. Popcorn numbers are always close to SMP Linux numbers, and for low core counts Popcorn performs better. In the best case (on 2 and 4 cores running CG), it is more than 50% faster than SMP Linux. In the worst case (on 16 cores running FT), Popcorn is 30% slower the SMP Linux.

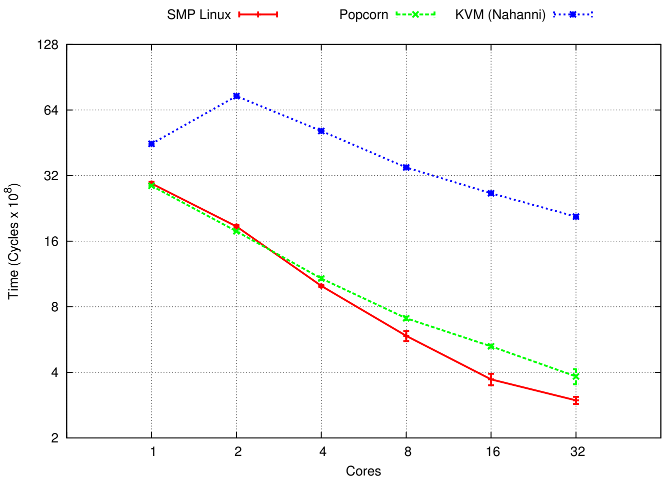

Figure 9: NPB/MPI integer sort (IS) benchmark results.

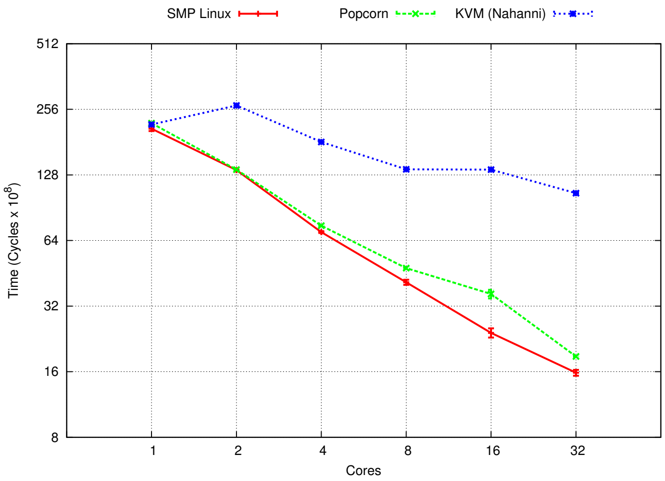

Figure 10: NPB/MPI fast Fourier transform (FT) benchmark results.

In Linux, MPI runs a process per core, but such processes are not pinned to the cores on which they are created: they can migrate to another less loaded core if the workload becomes unbalanced. In Popcorn, each process is pinned by design on a kernel running on a single core; if the load changes, due to system activities, the test process can not be migrated anywhere else. This is part of the cause of the observed SMP Linux's trends. Popcorn must also pay additional overhead for virtual networking and for running the replicated-kernel OS environment. We believe that communication via the virtual network switch is the main source of overhead in Popcorn. Considering the graphs, this overhead appears to be relatively small, and in general, the numbers are comparable. Finally, Popcorn enforced isolation results in a faster execution at low core counts; this shows that Linux's distributed work stealing scheduling algorithm can be improved.

Nahanni is the worst in terms of performance. It is up to 6 times slower than SMP Linux and Popcorn (on 32 cores running either CG or FT). Although Nahanni's one core performance is the same as SMP Linux, increasing the core count causes the performance to get worse on all benchmarks. A slower benchmark execution was expected on Nahanni due to the overhead incurred by virtualization. However, the high overhead that was observed is not just because of virtualization but is also due to inter-VM communication and scheduling on the Linux host.

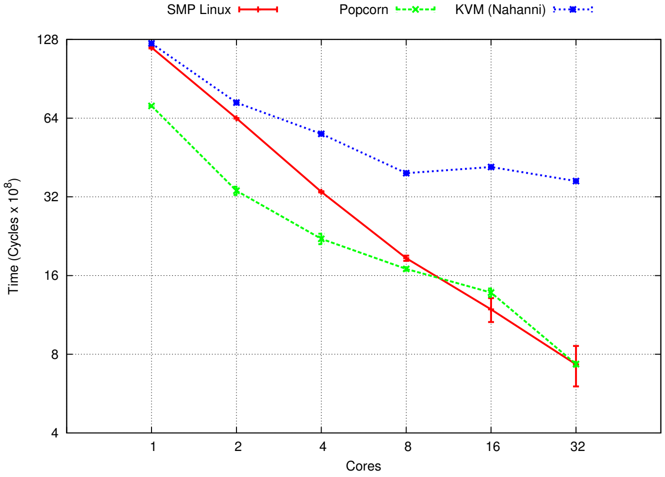

Figure 11: NPB/MPI conjugate gradient (CG) benchmark results.

OpenMP Figures 12, 13 and 14 show how Popcorn and SMP Linux perform on IS, FT, and CG benchmarks, respectively, within OpenMP. Graphs are in log_2 scale. For each data point, we ran 20 iterations, and we provide average and standard deviation.

Unlike the MPI experiments, there is no obvious common trend among the experiments. In the IS experiment, Popcorn is in general faster than SMP Linux, SMP Linux is up to 20% slower (on 32 cores). The FT experiment shows similar trends for Popcorn and SMP Linux, although SMP Linux is usually faster (less than 10%) but not for high core counts. In the CG experiment Popcorn performs poorly being up to 4 times slower than SMP Linux on 16 cores.

These experiments show that the performance of Popcorn depends on the benchmark we run. This was expected, as our address space consistency protocol performance depends on the memory access pattern of the application. Analysis and comparison of the overhead in SMP Linux and Popcorn Linux reveals that the removal of lock contention from SMP Linux yields significant gains for Popcorn over SMP Linux. This contention removal is an artifact of the fact that data structures are replicated, and therefore access to those structures does not require significant synchronization. However, Popcorn must do additional work to maintain a consistent address space. These two factors battle for dominance, and depending on the workload, one will overcome the other. Mechanisms are proposed to reduce the Popcorn overhead with the goal of making further performance gains on SMP Linux.

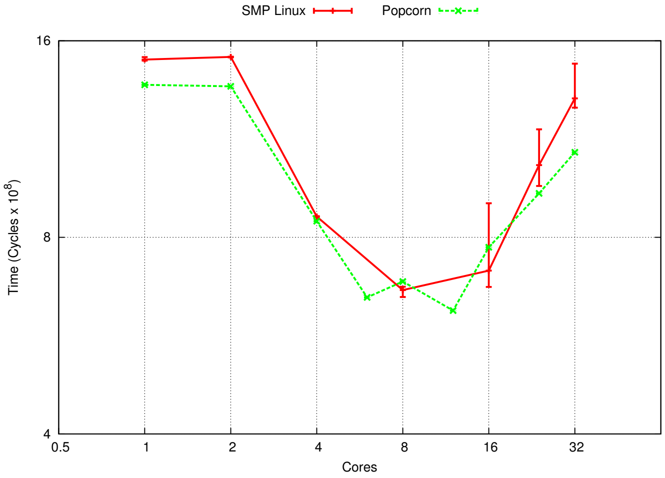

Figure 12: NPB/OpenMP integer sort (IS) benchmark results.

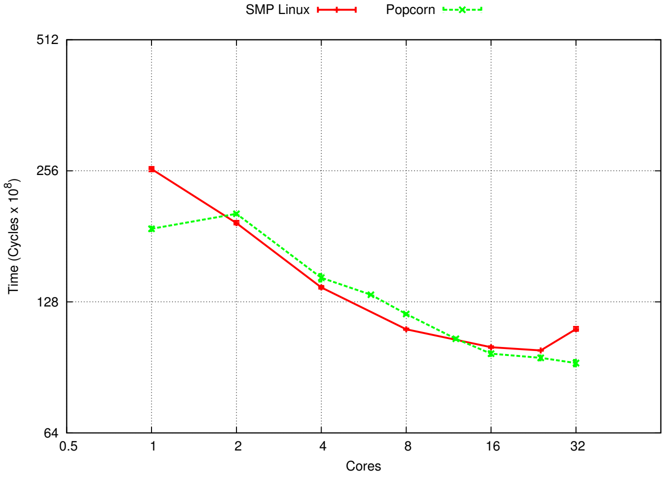

Figure 13: NPB/OpenMP fast Fourier transform (FT) benchmark results.

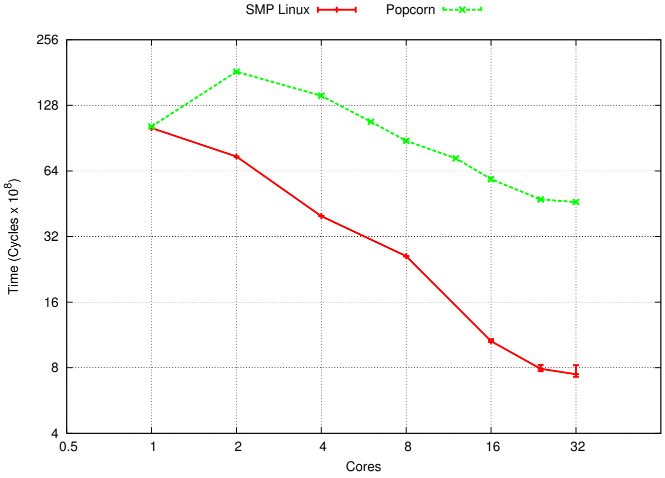

Figure 14: NPB/OpenMP conjugate gradient (CG) benchmark results.

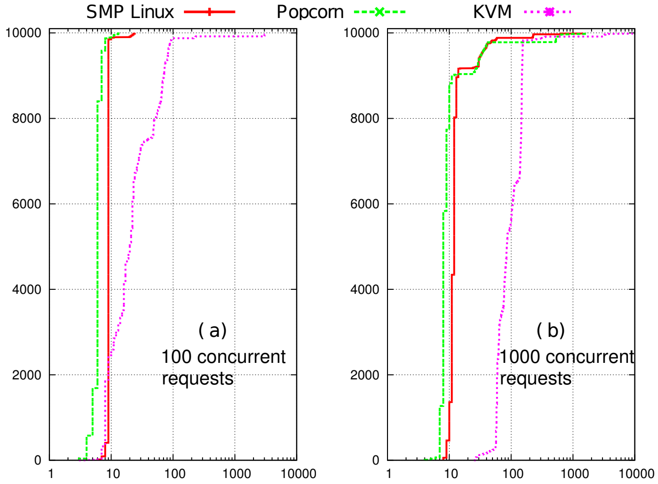

Figure 15: *Apache Bench* results on *nginx* web server on SMP Linux, Popcorn and KVM. Response time in *ms* on the *x* axis, number of requests on the *y* axis.

6.2 I/O Bound

We set ApacheBench to run 10000 requests with a concurrency level of 100 and 1000 threads to obtain the data in Figure 15.

The data show that the Popcorn architecture does not hinder the performance of running a web server (I/O bound benchmark) on the selected hardware, when compared to SMP Linux. Figure 15 shows that in most cases, Popcorn can serve a request faster then Linux, which is attributable both to scheduling and to the fact that the task of handling the hardware network interface is shared with the kernel instance that owns it. KVM suffers due to virtualization and scheduling overhead.

7 Conclusions

We introduced the design of Popcorn Linux and the re-engineering required by the Linux kernel to boot a coexistent set of kernels, and make them cooperate as a single operating system for SMP machines. This effort allowed us to implement and evaluate an alternative Linux implementation while maintaining the shared memory programming model for application development.

Our project contributes a number of patches to the Linux community (booting anywhere in the physical address space, task migration, etc.). Popcorn's booting anywhere feature offers insight into how to re-engineer Linux subsystems to accommodate more complex bootup procedures. Task migration enables Linux to live migrate execution across kernels without virtualization support. Next steps include completing and consolidating the work on namespaces, using rproc/rpmsg instead of *kexec* and re-basing the messaging on *virtio*.

Our MPI results show that Popcorn provides results similar to Linux, and that it can outperform virtualization-based solutions like Nahanni by up to a factor of 10. The network test shows that Popcorn is faster than Linux and Nahanni. Popcorn is not based on hypervisor technologies and can be used as an alternative to a set of virtual machines when CPUs time-partitioning is not necessary. The OpenMP results show that Popcorn can be faster, comparable to, or slower then Linux, and that this behaviour is application dependent. The replicated-kernel OS design applied to Linux promises better scaling, but the gains are sometimes offset by other source of overheads (e.g. messaging). An analysis on a thousands of cores machine can provide more insight into this solution.

It turns out that a replicated-kernel OS based on Linux aggregates the flexibility of a traditional single-image OS with the isolation and consolidation features of a virtual machine, but on bare metal, while being potentially more scalable on high core count machines. The full sources of Popcorn Linux and associated tools can be found at http://www.popcornlinux.org.

Acknowledgments

This work would not have been possible without the help of all the people involved in the Popcorn project. We are particularly thankful to Ben Shelton, Sean Furrow, Marina Sadini, Akshay Ravichandran, Saif Ansary, and Alastair Murray for their contributions. Moreover we are thankful to Cam MacDonell at the University of Alberta for helping us install the Nahanni code and giving us access to their private source code.

References

[1] An introduction to the Intel quickpath interconnect. 2009.

[2] Technical advances in the SGI UV architecture, 2012.

[3] Samsung exynos 4 quad processor, 2013.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[6] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[7] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, July 2012.

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[9] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10 pp. –530, may 2006.

[10] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, Washington, DC, USA, 2009. IEEE Computer Society.

[11] John Chapin and et al. Hive: Fault containment for shared-memory multiprocessors, 1995.

[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[13] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16 –29, march-april 2010.

[14] Vivek Goyal. ELF relocatable x86 bzimage (v2), 2006.

[15] Intel Corporation. Xeon Phi product family. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[16] A. Kale, P. Mittal, S. Manek, N. Gundecha, and M. Londhe. Distributing subsystems across different kernels running simultaneously in a Multi-Core architecture. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 114–120, 2011.

[17] Xiaodi Ke. Interprocess communication mechanisms with Inter-Virtual machine shared memory. Master's thesis, University of Alberta, August 2011.

[18] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. Towards an efficient single system image cluster operating system. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 370–377, Oct 2002.

[19] Yoshinari Nomura, Ryota Senzaki, Daiki Nakahara, Hiroshi Ushio, Tetsuya Kataoka, and Hideo Taniguchi. Mint: Booting multiple linux kernels on a multicore processor. pages 555–560. IEEE, October 2011.

[20] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and

Phil Winterbottom. Plan 9 from bell labs. *Computing Systems*, 8(2):221–254, 1995.

[21] T. Shimosawa, H. Matsuba, and Y. Ishikawa. Logical partitioning without architectural supports. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 355–364, July 2008.

[22] Ron Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9:105–134, 1993.

[23] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[24] Karim Yaghmour. A practical approach to linux clusters on smp hardware. Technical report, 2002.