Towards Operating System Support for Heterogeneous-ISA Platforms

Antonio Barbalace, Alastair Murray, Rob Lyerly, Binoy Ravindran
Dept. of Electrical and Computer Engineering
Virginia Tech, Virginia, USA
{antoniob, alastair, rlyerly, binoy}@vt.edu

ABSTRACT

Given an emerging trend towards OS-capable heterogeneous-ISA multi-core processors, we address the problem of how to redesign classic symmetric multi-processing (SMP) operating systems (OS) to exploit this hardware. We propose an OS design that consists of multiple kernels, each one compiled for, and run on, a specific ISA of the heterogeneous platform. These kernels collaboratively maintain a distributed OS state, share hardware resources and transfer their workload. Following these design principles, we identify a set of features that should be implemented in SMP OSs to realize our OS design.

We deploy these features in Linux to produce a homogeneous prototype of our OS design. We evaluate this prototype by partitioning a multi-processor machine to run multiple kernels. We compare against traditional Linux to demonstrate that our redesign does not hinder performance.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

Keywords

Many-core, Heterogeneous, Operating-Systems, Linux

1. INTRODUCTION

The trend towards parallel hardware in computer systems is well established. This trend covers not only massively-parallel accelerators such as GPUs, but also general-purpose OS-capable many-core processors. More recently, a second trend has begun to develop where OS-capable multi-core processors are also becoming more diverse and heterogeneous. While classic symmetric multi-processing (SMP) operating systems (OS) have been able to scale to many-core hardware with some re-design [3], further re-structuring will be required to support more diverse hardware. This paper proposes such a re-structuring and demonstrates its validity with an initial implementation based on the Linux kernel.

1.1 Motivation

The parallelism trend has progressed far enough that in many contexts, single-core processors are rare; multi-core processors have become standard hardware. Different levels of parallelism in hardware designs result in different performance profiles. Multi-core processors have a few large processing units, whereas many-core processors are made up from many small processing units. This in turn means that multi-core processors have high per-core performance and moderate-throughput but high-power demands, while many-core processors have lower per-core performance but high-throughput with low-power demands (but only on parallel workloads).

Many-core processors such as the Intel Xeon-Phi (60 processors, 240 threads) or the Tilera TILE-Gx72 (72 processors) are commercially available; both have complete operating system support and can run Linux. However, in an attempt to provide good performance in a large range of situations, the latest generation of processor designs integrate different types of processing units onto a single die. The Intel Sandy Bridge processor has up to six CPU cores and up to twelve GPU cores. ARM has developed the big.LITTLE architecture, which combines a high-performance four-core processor with a low-power four-core processor in order to handle high-performance workloads without wasting power on low-demand workloads.

While increasing the diversity of processors enables better performance (e.g. scheduling applications for performance or power, load balancing across processors, etc.), modern systems only loosely integrate OS-capable hardware by concurrently executing separate OSs within the same system. Without a single namespace and coherent system state, applications must be rewritten to explicitly take advantage of the performance benefits of multiple-ISA systems. However, this breaks the classic OS/process abstraction model and prevents the OS from providing many services (e.g. load balancing and process migration for device locality). Clearly, an OS that provides a single system image (SSI) and transparently handles hardware diversity has many benefits.

These trends towards diversity and parallelism, and the lack of complete OS support for such systems, inform our design for a messaging-based multiple kernel OS that supports heterogeneous-ISA architectures, while hiding hardware diversity from applications.

1.2 Contribution

In this paper we propose a redesign of the traditional SMP OS in order to support future heterogeneous hardware. To demonstrate its feasibility we describe a prototype that partitions an x86 multi-core machine into either one kernel per core or one per multi-core processor. We apply this heterogeneous design to Linux to take advantage of the mature ecosystem that has developed around traditional SMP operating systems. Our contributions are:

- We introduce inter-kernel communication to maintain a global OS state amongst different kernel instances that may be running on different ISAs.

- We propose a transparently managed client/server model for allowing inter-kernel access to local resources.
- In our prototype kernel, we model a minimum set of subsystems that must be kept coherent in order for a heterogeneous OS to work. From this analysis we identify a set of features that must be supported by the OS.
- We deploy such features in Linux, and we evaluate their functionality on homogeneous-ISA as a working prototype of our model. Our prototype is called *Popcorn Linux*, and the sources are publicly available.

2. RELATED WORK

To the best of our knowledge we are the first to present a redesign of Linux to accommodate heterogeneous-ISA hardware, but there is previous work in both general heterogeneous and multiple kernel operating systems.

Li et al. [8] deploy a Linux design for overlapping-ISA heterogeneous architectures. In their model cores share a large set of common instructions and registers with identical encoding and semantics; we target a much broader hardware model. We know of three separate Linux-based efforts that have implemented a partitioned design to target homogeneous x86 hardware: SHIMOS [6], Twin Linux [7] and Linux Mint [10]. The purpose of these works, however, was not to address heterogeneous hardware. Further, Twin Linux only supports shared-memory inter-kernel communication, while Linux Mint and SHIMOS do not support any method of communication at all.

Most of the research in the area of heterogeneous OSs involves the creation of new kernels. The disadvantage of this approach is the loss of support and compatibility that Linux provides. The approach that is the most similar to our own is the Barrelfish OS [2]. This is the multikernel design that runs a microkernel per-core. Barrelfish uses message passing to keep global system state coherent, in a manner resembling a closely-coupled distributed system. A heterogeneous version of the Barrelfish model has also been proposed [12]. Although we use a similar message passing design, we do not use RPC; instead, we use task-to-task messaging. Furthermore our primary goal is to handle diversity. The Helios OS [9] is entirely designed around heterogeneity, also using a multikernel-like design on .NET. In contrast to our design, however, kernels are not peers; instead there are designated coordinator and satellite kernels.

Finally, kernel-level scheduling approaches to support overlapping-ISA heterogeneous architectures have been proposed for x86 [13], where an application can run on any processor but will experience different performance per processor.

3. OS DESIGN PRINCIPLES

The traditional SMP OS cannot deal with emerging heterogeneous-ISA processors, but these monolithic OSs are very widely used and have an enormous level of software support. A microkernel design may seem a more natural fit when constructing a single OS out of multiple kernels (i.e. the multikernel), but we prefer to follow a more traditional design so as to maintain as much compatibility with existing software as is possible. We attempt to provide a single system image so that applications are not aware that they may be running across multiple kernels, but the OS exports the system topology so that applications may choose an optimal mapping. Of course, providing this single system image requires several changes and additions to the monolithic kernel model – these changes are described below.

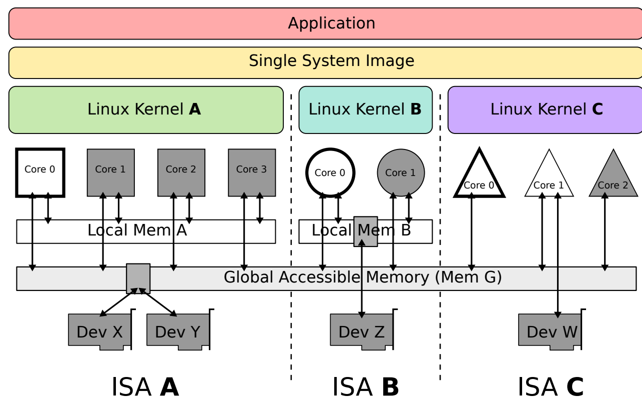

Figure 1: Heterogeneous-ISA hardware model.

A heterogeneous OS consists of multiple kernels that are compiled for different ISAs. While multiple kernel instances run together on the various ISAs, their state must be kept coherent in order to provide applications with the view of a single OS; this is accomplished by partially replicating global OS state. We assume that not every kernel object's state must be replicated on each kernel, but that only the kernels that are accessing a specific replicated object must know about any state changes (note that an object can be a task, an address space, etc.). This allows us to reduce inter-kernel communication and synchronization.

3.1 Hardware Model

In the following, we assume a hardware model in which processors of different ISAs share access to a global, eventually consistent, memory (*Mem G* in Figure 1). We also allow hardware message passing. Computational units of a single ISA could have exclusive access to a memory area (*Mem A* and *Mem B*) and across ISAs the same memory area can be mapped at different physical address ranges. A similar model holds for accessing devices and peripherals that are mainly memory-mapped. Some devices, like *Dev X* and *Dev Y*, can be directly accessed by any processor. Others, like *Dev Z* or *Dev W*, cannot. The schema in Figure 1 is representative of the hardware model that our heterogeneous OS design could support.

In the introduction we described current market trends and future hardware possibilities, but Figure 1 is representative of present-day hardware. Limitations of current hardware, however, introduce the caveat that heterogeneous memory accesses, such as *ISA C* accessing *Mem G*, are not cache coherent. It represents a loosely-coupled heterogeneous system connected via the PCIe bus, e.g. a multi-core x86 machine (*ISA C*) with Tilera TILE-Gx72 (*ISA B*) and Intel Xeon-Phi (*ISA A*) boards.

3.2 Peer Kernels

In a setup where different kernels coexist, a relationship must be defined between them. In virtual machine environments, for example, the OS running on the bare hardware is called the *host OS*, and OSs running on the virtual machines are called *guest OSs*. The relationship is hierarchical: guest

OSs are nested into the host OS, which provides services to them. In our approach, all kernel instances are peers that reside on processors of different ISAs. In a peer relationship, kernels do not necessarily depend on the others for services; from any kernel, it will be possible to boot any other kernel, run any service, and control any of the hardware devices, if they are physically accessible from the processing units on which a kernel is running.

Physical hardware design will, however, imply some level of hierarchy in some circumstances. For example with an x86 motherboard and an OS-capable PCIe device, it is unlikely that the PCIe device will be able to boot kernels on x86 cpus. As a consequence of the boot process of the multi-processor x86 architecture the kernel is launched by the BIOS/boot loader on a *bootstrap processor*. After some basic initialization, it sequentially starts all the other CPUs in the system (called *application processors*). In the hardware model described, we foresee that a single bootstrap processor, which belongs to an ISA such as x86, will be booted first, and it will be responsible for booting all other processors. The first kernel to boot on a machine is the *Primary* kernel, and all others are called *Secondary* kernels.

Required Features. Kernels must be able to boot other kernels and to provide any service. In order for different kernel instances to be peers, they must be able to communicate. Communication enables coordination and system state consistency across peer kernels such that all peers form a single operating system. As shown in our hardware model, within a heterogeneous ISA platform different processors can potentially be connected via different buses. The communication layer should be flexible enough to exploit the optimal method of providing fast and reliable communication between kernels on the given hardware, including shared memory and hardware message passing.

3.3 Resource Sharing

In a heterogeneous platform, computational units, memories, accelerators, and peripherals of many types may be available. In a heterogeneous OS resources are allocated per kernel instance. Computational units do not always have direct access to all hardware resources; therefore, each kernel may not be able to directly access every device in a platform. A heterogeneous OS should be able to hide this diversity and allow each kernel access every resource present in a platform. When hiding is too expensive, the application or part of it should be migrated closer to the resource (see Section 3.4).

Global Resources. Globally physical accessible resources are hardware resources that can be seen by each computational unit, like *Mem G*, *Dev X*, and *Dev Y* in Figure 1. A global resource, or part of it, can be accessed concurrently by different kernels; for example, a chunk of a global shared memory (*Mem G*) can be used for communication or synchronization by all the kernels. Otherwise, globally accessible resources should be partitioned amongst kernels. In the process of selecting which kernel instance owns which subset of global resources, different criteria should be evaluated (e.g. proximity of a resource to a computational unit).

Globally accessible resources may not be partitionable to all kernel instances if they are limited in number (like *Dev X* and *Dev Y*). In this case, to provide access to a single indivisible resource amongst different kernels, a resource can be time-shared between kernels, or the kernel that owns the resource can act as a server to let the others access that resource. In the time sharing case, if a resource is initially assigned to a kernel, its ownership will be changed at runtime and all the kernels will be notified. If a single kernel owns the device, the other kernels interact with the device by communicating with the owner kernel that acts as a server or proxy to that device.

Local Resources. Local resources are hardware resources that are not physically accessible by all computational units, but only by a subset of them. In Figure 1 *Mem A*, *Mem B*, *Dev Z*, and *Dev W* are only directly accessible by processors of a specific ISA. The only way to deal with such resources is to use the client/server model, where a server kernel can directly access the resource and provide the services to all the other kernels.

Required Features. A kernel must be able to initialize all devices local to that ISA, and make them accessible by proxy. To enable sharing of global resources, inter-kernel communication is again required to allow coordination between kernels requesting the same resource. A replicated state of the current allocation of each hardware resource must be maintained across kernels, and every kernel must know if access to a particular resource is proxied by another kernel. From an application point of view, resource access must be transparently managed by the OS, and the application should see a consistent resource namespace on each kernel instance.

3.4 Load Sharing

As in cluster environments, different kernels should share their workload. Note that remote process migration across heterogeneous ISAs is out of the scope of this article and is not implemented in our homogeneous prototype. We intend that cross-ISA execution migration would work in a manner similar to that proposed by DeVuyst et al. [5]. They assume a single OS running on different cores and use a combination of multiple-compilation and dynamic binary translation to allow efficient heterogeneous-ISA execution migration.

The advantage of remotely creating and migrating processes and threads is not only to provide load balancing across ISAs, but also to realize power saving policies, to improve latency when accessing resources local to a particular kernel, and to exploit the faster execution of a particular piece of code on a specific ISA.

Required Features. To migrate applications across kernels, a single system image is required in order for them to execute correctly. A single and consistent view of the filesystem must be provided as well. For migrated applications to communicate or synchronize with applications resident on other kernels, inter-kernel inter-process communication primitives should be provided. This in turn requires functionalities, like a single process identification space, to be maintained across kernels. At each application's migration, interested kernels must be informed. Inter-kernel scheduling must exploit inter-kernel communication and must happen collaboratively amongst different kernels.

4. X86 IMPLEMENTATION

We deployed the presented design in Linux 3.2.14, modifying 159 files while adding a total of $\sim 29k$ lines. We tested its functionality on homogeneous x86 multi-core hardware, but to demonstrate heterogeneous support we treat the homogeneous hardware as if it were heterogeneous. Every core, or group of cores, runs a single-ISA kernel.

4.1 Boot Process

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 within the kernel to load a new kernel instance that will run in parallel on a different partition of hardware resources ($\sim 4k$ lines). Following our design, the primary kernel is loaded on some combination of processors that, by design, includes the bootstrap processor. Each secondary kernel is loaded on some group of application processors that are not in use by any other kernel instance. Processors are not enumerated as in Linux but are given globally unique IDs that make them identifiable across all kernel instances.

In order to boot each secondary kernel instance, we have created a modified version of the trampoline[1] used to boot application processors in SMP Linux. We rewrote the kernel bootup code in order to boot kernels at locations throughout the physical address space[2] (we currently support only the x86-64 architecture).

4.2 Shared Memory Handling

We divided the shared memory into chunks, some of which are private to a single kernel and some of which are shared between kernels. Private chunks, or sub-chunks, can be shared after boot up with private shared mappings.

Linux already offers functionalities to start a kernel with a reduced view of the available physical memory. We use such functionalities to statically allocate private chunks to each kernel. In order to provide for runtime private shared mappings we borrow `remap_pfn_range` and `ioremap_cache`. These mappings are managed by a memory service running on each kernel.

4.3 Device Drivers

A redesign of the device initialization was necessary to allow a secondary Linux kernel to boot. We implemented a resource-masking feature to initialize only the devices owned by a kernel, and we also had to check device drivers for their hardware discovery code. User-space drivers must always run on the kernel local to their device.

The I/O APIC, the programmable interrupt controller on the x86 architecture, is a tangible example of a global resource that must be accessed exclusively by one kernel at a time. We developed a server driver to proxy accesses to the APIC from other kernels. Currently, the I/O APIC driver is loaded exclusively on the primary kernel, and any other kernels with a device that requires interrupts in their resource partition will communicate with the primary kernel in order to operate with the I/O APIC. Conversely, the CD-ROM device is an example of time-sharable device. E.g., a "CD Burner" application must have exclusive access to the CD-ROM device for the time it takes to write a disc.

4.4 Kernel to Kernel Communication

To communicate between kernel instances, we developed a kernel-level, pluggable, message passing framework. We address here our plugin that exploits shared memory and an hybrid of polling and IPIs (inter-processor interrupts). We chose a pluggable approach to ease portability and to support multiple communication channels between kernels. The framework provides priority-based synchronous and asynchronous messaging. The format, priority, and synchronicity of a message is kernel object dependent.

[1]`arch/x86/kernel/trampoline_64.S`
[2]`arch/x86/kernel/head_64.S`

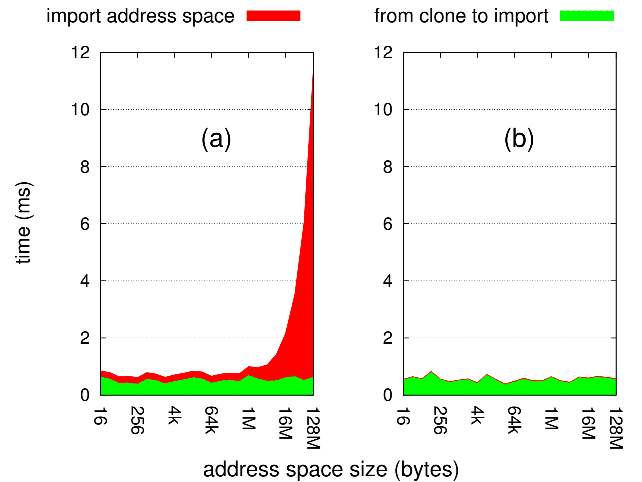

Figure 2: Process migration in Popcorn. The time to import the address space information of the migrated process varying the size of its heap. Figure (a) migrates a process's entire address space at once whereas Figure (b) uses on-demand page migration.

Shared Memory Plugin. The shared-memory plugin uses cache-aligned private buffers over memory that is mapped into each kernel's address space. For unicast messaging, each kernel allocates its own buffer at boot time and then makes this physical address available to the other kernels. For multicast we introduced multicast groups that allow messages to be passed to all members in the group with a single write into a shared buffer. Multicast groups are opened and closed dynamically at runtime; we are currently using them to keep a distributed process address space consistent across kernels.

To notify a kernel that a message was sent to it we send an IPI. IPIs on the x86 architecture have non-negligible overhead. To limit this effect we reduced the number of IPIs by disabling interrupts when processing a batch of messages. This allows high-bandwidth communication during periods of high-demand, while providing low-overhead during periods of low-demand.

4.5 Task Migration

To migrate processes or threads between kernel instances, a client/server model has been adopted. When the user or the scheduler triggers an inter-kernel task migration, a migration service on the remote kernel is contacted. We have described thread migration in previous work [11]; here we consider process migration. From a Linux stand point, we assume that no OS-level resources are shared between processes. To support remote task creation we added a further argument to `sys_clone` in order to specify on which kernel start executing the task.

Process Migration. An inter-kernel process migration comprises of the following steps: firstly, the local process which is to be migrated is stopped. Secondly, all the relevant information about the process is transferred to the server where a dummy process that acts as the migrated process is started on the remote kernel. Finally, all the transferred informa-

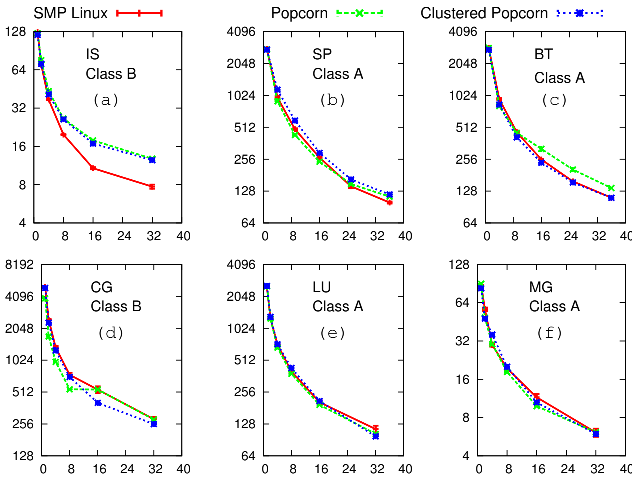

Figure 3: NPB class B benchmarks in the first column. NPB class A (smaller problem size) benchmarks in the other columns. Each subgraph has time (in $cycles * 10^8$) on the y-axis and *core count* on the x-axis.

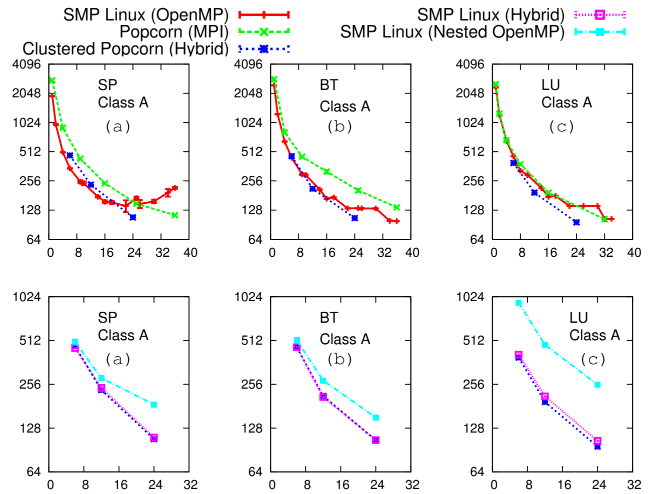

Figure 4: A comparison of the "application" benchmarks in NPB on Popcorn and SMP Linux using various programming models. The top line compares the most natural programming model for each OS. The code for Clustered Popcorn comes from NPB-MZ, so the bottom line compares NPB-MZ on both Clustered Popcorn and SMP Linux.

tion about the migrated process is imported into the dummy process and the process is ready to continue executing.

We use the messaging layer to transfer the process state between kernels. The state of the process is comprised of the contents of all registers and its address space. An address space is comprised of virtual memory area information, and the map of physical pages to those virtual memory areas. On homogeneous platforms in order to avoid copying memory contents, we only reproduce these virtual memory area to physical memory mappings in the receiving kernel on behalf of the newly migrated process. To account for the memory used by each migrated process we leave a shadow process on each kernel where the process allocated memory.

We benchmarked this process, measuring the time required to migrate a process in Popcorn by varying the size of the program's address space. The results are shown in Figure 2(a) for upfront migration of the address space, and in Figure 2(b) for the on-demand counterpart. These results exclude messaging time, as this varies with hardware and memory topology. `sched_setaffinity()` on SMP Linux takes $0.6ms$ on average. Importing the address space on the remote kernel in Linux (installing `struct_vms` and mapping `pte_structs`) requires a time proportional to the size of the address space (red area). A task can be migrated on-demand in $6ms$ (messaging included).

4.6 Software Network Switch

As is common practice in virtualization, we provide software networking between different kernel instances. In our implementation, the kernel instance that owns the network card acts as the gateway machine and routes traffic to and from the other kernel instances. In this setup, each kernel instance has an associated IP address, and switching is automatically handled at the driver level. A network overlay 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 for notification and fast shared-memory ring buffers for communication. Our implementation uses the kernel-level messaging framework described earlier for coordination and check-in.

5. EVALUATION

The purpose of this evaluation is to demonstrate that the restructuring that we propose in Section 3 can be applied to Linux while maintaining functionality.

Though our design includes remote task creation and migration across processors of different ISAs, this is a difficult problem and the research space is quite sparse. For this reason, the following evaluation focuses on explicitly parallel benchmarks where tasks do not migrate from their kernels once created. We evaluated Popcorn using MPI versions of the NAS Parallel Benchmarks (NPB). We compare Popcorn to Linux to highlight any performance penalties that arise from coordinating multiple kernels.

Experimental Setup. We run the following experiments on a Supermicro H8QG6 equipped with four AMD Opteron 6164 processors, running at 1.7GHz, with 64GB of RAM for a total of 48 cores in a ccNUMA configuration. We report measurements on Popcorn configured as one kernel per core and one kernel per NUMA node (6 cores); we called these configurations *Popcorn* and *Clustered Popcorn* respectively. For all the experiments we used Linux 3.2.14 on which Popcorn is actively developed, and we adopt our version of `kexec` (based on version 2.0.3) to boot Popcorn's kernel instances.

These experiments investigate how our design impacts compute-bound workloads. We used the MPI versions of NASA's NAS Parallel Benchmark (NPB) suite [1], using the MPICH2 MPI library. We developed a MPI-Popcorn version of MPICH2 that uses the SSH interface for initial orchestration and process management, but modifies the Nemesis [4] sub-system to use inter-kernel shared-memory for inter-process communication. Although such benchmarks do not stress the OS, the `sys_sched_yield` and `sys_poll`

syscalls are heavily used. We consider these benchmarks sufficient to demonstrate that Popcorn Linux is functional and that the homogeneous prototype has acceptable performance, while also demonstrating interesting results for multiple kernel OSs.

Although we used a 48-core machine, the recorded results only go up to 36-cores because the NPB MPI benchmarks cannot run on an arbitrary number of cores. Some require a power-of-two, others a square number of cores. For each data point we ran 20 iterations and provide the average runtime. We chose small data sets (class A and B) because we wanted OS effects to be visible: longer-running processes are less influenced by the underlying kernel.

Results. Figure 3 compares the performance of Popcorn (1-core per kernel), Clustered Popcorn (6-cores per kernel) and SMP Linux (a single 48-core kernel). For high core counts, Popcorn is slightly slower than SMP Linux; this is particularly visible in the IS graph (Figure 3(a)). At low core-counts, however, Popcorn can experience a slight performance advantage (e.g. in Figure 3(d)) as each kernel only has one processor, so cache and NUMA locality are enforced and the scheduler cannot move a process to a different processor. Similarly, every processor in a Clustered Popcorn kernel belongs to a single NUMA-zone and share an L3 cache. Popcorn's reduced performance at high core counts is from the overhead of having a kernel per core. Having six cores per kernel in Clustered Popcorn is enough to eliminate this overhead.

Finally, as OpenMP is a more natural method of writing parallel programs in SMP Linux than MPI, we compare the performance of this against Popcorn in Figure 4. As this is comparing programming models, we only consider the application benchmarks within NPB; these are also the benchmarks available in the multi-zone version of NPB, thus allowing us to try hybrid MPI/OpenMP models. The top row of Figure 4 shows that OpenMP on SMP Linux is, predictably, faster than MPI on Popcorn, though the OpenMP version of SP has scaling issues. The hybrid MPI/OpenMP code on Clustered Popcorn, however, out-performs both. The bottom line of Figure 4 shows that running the same hybrid code on SMP Linux has almost identical performance, though the nested OpenMP implementation is slower. Thus, Clustered Popcorn is as efficient as SMP Linux for the optimal versions of these benchmarks.

6. CONCLUSION

In this paper, we have begun to address the problem of how to re-design a traditional SMP OS to allow it to run on heterogeneous-ISA hardware. We did this while extending common design principles that are found in SMP OS design, such as resource sharing. We claim that an OS running on a heterogeneous-ISA platform should be made up of different kernel instances, compiled for each ISA, but with coherent and replicated OS state. We deployed the necessary features to realize our design principles in Linux and found that they do not hinder performance when compared to SMP Linux.

As future work, we would like to deploy Popcorn Linux on genuine heterogeneous ISA hardware. We will explore how to exploit such platforms, not only to improve the coordinated usage of hardware resources, but also to improve performance by exploiting the heterogeneous nature of these platforms. This will likely require compiler-based modifications such that a program can interact with the OS scheduler to allow optimal mapping.

The full sources for Popcorn Linux and associated tools can be found at `http://www.popcornlinux.org`

7. ACKNOWLEDGMENTS

This work is supported by the US Office of Naval Research under Contract N00014-12-1-0880.

References

[1] 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 *Supercomputing '91*, 1991.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, 2009.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. OSDI'10, 2010.

[4] G. M. D. Buntinas and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. 2006.

[5] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In *ASPLOS XVII*, 2012.

[6] T. Himosawa, H. Matsuba, and Y. Ishikawa. Logical partitioning without architectural supports. In *COMPSAC '08*.

[7] A. Kale, P. Mittal, S. Manek, N. Gundecha, and M. Londhe. Distributing subsystems across different kernels running simultaneously in a Multi-Core architecture. In *CSE XIV*, 2011.

[8] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA '10*, 2010.

[9] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *SOSP '09*, 2009.

[10] Y. Nomura, R. Senzaki, D. Nakahara, H. Ushio, T. Kataoka, and H. Taniguchi. Mint: Booting multiple linux kernels on a multicore processor. In *BWCCA '11*, 2011.

[11] M. Sadini, A. Barbalace, B. Ravindran, and F. Quaglia. A page coherency protocol for popcorn replicated-kernel operating system. In *MARC V*, 2013.

[12] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *MMCS '08*, 2008.

[13] S. Srinivasan, R. Iyer, L. Zhao, and R. Illikkal. HeteroScouts: Hardware Assist for OS Scheduling in Heterogeneous CMPs. In *SIGMETRICS '11*, 2011.