Decoupling Cores, Kernels, and Operating Systems

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe, *ETH Zürich*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zellweger

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Decoupling Cores, Kernels, and Operating Systems

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

Abstract

We present Barrelfish/DC, an extension to the Barrelfish OS which decouples physical cores from a native OS kernel, and furthermore the kernel itself from the rest of the OS and application state. In Barrelfish/DC, native kernel code on any core can be quickly replaced, kernel state moved between cores, and cores added and removed from the system transparently to applications *and* OS processes, which continue to execute.

Barrelfish/DC is a multikernel with two novel ideas: the use of *boot drivers* to abstract cores as regular devices, and a partitioned capability system for memory management which *externalizes core-local kernel state*.

We show by performance measurements of real applications and device drivers that the approach is practical enough to be used for a number of purposes, such as online kernel upgrades, and temporarily delivering hard real-time performance by executing a process under a specialized, single-application kernel.

1 Introduction

The hardware landscape is increasingly dynamic. Future machines will contain large numbers of heterogeneous cores which will be powered on and off individually in response to workload changes. Cores themselves will have porous boundaries: some may be dynamically fused or split to provide more energy-efficient computation. Existing OS designs like Linux and Windows assume a static number of homogeneous cores, with recent extensions to allow core hotplugging.

We present Barrelfish/DC, an OS design based on the principle that all cores are fully dynamic. Barrelfish/DC is based on the Barrelfish research OS [5] and exploits the "multikernel" architecture to separate the OS state for each core. We show that Barrelfish/DC can handle dynamic cores more flexibly and with far less overhead than Linux, and also that the approach brings additional benefits in functionality.

A key challenge with dynamic cores is safely disposing of per-core OS state when removing a core from the system: this process takes time and can dominate the hardware latency of powering the core down, reducing any benefit in energy consumption. Barrelfish/DC addresses this challenge by externalizing all the per-core OS and application state of a system into objects called *OSnodes*, which can be executed lazily on another core. While this general idea has been proposed before (notably, it is used in Chameleon [37] to clean up interrupt state), Barrelfish/DC takes the concept much further in *completely* decoupling the OSnode from the kernel, and this in turn from the physical core.

While transparent to applications, this new design choice implies additional benefits not seen in prior systems: Barrelfish/DC can completely replace the OS kernel code running on any single core or subset of cores in the system at runtime, without disruption to any other OS or application code, including that running on the core. Kernels can be upgraded or bugs fixed without downtime, or replaced temporarily, for example to enable detailed instrumentation, to change a scheduling algorithm, or to provide a different kind of service such as performance-isolated, hard real-time processing for a bounded period.

Furthermore, per-core OS state can be moved between slow, low-power cores and fast, energy-hungry cores. Multiple cores' state can be temporarily aggregated onto a single core to further trade-off performance and power, or to dedicate an entire package to running a single job for a limited period. Parts of Barrelfish/DC can be moved onto and off cores optimized for particular workloads. Cores can be fused [26] transparently, and SMT threads [29, 34] or cores sharing functional units [12] can be selectively used for application threads or OS accelerators.

Barrelfish/DC relies on several innovations which form the main contributions of this paper. Barrelfish/DC treats a CPU core as being a special case of a peripheral device, and introduces the concept of a *boot driver*, which can start, stop, and restart a core while running elsewhere. We

use a *partitioned capability system* for memory management which allows us to completely externalize all OS state for a core. This in turn permits a kernel to be essentially stateless, and easily replaced while Barrelfish/DC continues to run. We factor the OS into per-core kernels[1] and *OSnodes*, and a *Kernel Control Block* provides a kernel-readable handle on the total state of an OSnode.

In the next section, we lay out the recent trends in hardware design and software requirements that motivate the ideas in Barrelfish/DC. Following this, in Section 3 we discuss in more detail the background to our work, and related systems and techniques. In Section 4 we present the design of Barrelfish/DC, in particular the key ideas mentioned above. In Section 5 we show by means of microbenchmarks and real applications (a web server and the PostgreSQL database) that the new functionality of Barrelfish/DC incurs negligible overhead, as well as demonstrating how Barrelfish/DC can provide worst-case execution time guarantees for applications by temporarily isolating cores. Finally, we discuss Barrelfish/DC limitations and future work in Section 6, and conclude in Section 7.

2 Motivation and Background

Barrelfish/DC fully decouples cores from kernels (supervisory programs running in kernel mode), and moreover both of them from the per-core state of the OS as a whole and its associated applications (threads, address spaces, communication channels, etc.). This goes considerably beyond the core hotplug or dynamic core support in today's OSes. Figure 1 shows the range of primitive kernel operations that Barrelfish/DC supports transparently to applications and without downtime as the system executes:

- A kernel on a core can be rebooted or replaced.
- The per-core OS state can be moved between cores.
- Multiple per-core OS components can be relocated to temporarily "share" a core.

In this section we argue why such functionality will become important in the future, based on recent trends in hardware and software.

2.1 Hardware

It is by now commonplace to remark that core counts, both on a single chip and in a complete system, are increasing, with a corresponding increase in the complexity of the memory system – non-uniform memory access and multiple levels of cache sharing. Systems software, and in particular the OS, must tackle the complex problem of scheduling both OS tasks and those of applications across a number of processors based on memory locality.

At the same time, cores themselves are becoming non-uniform: Asymmetric multicore processors (AMP) [31] mix cores of different microarchitectures (and therefore performance and energy characteristics) on a single processor. A key motivation for this is power reduction for embedded systems like smartphones: under high CPU load, complex, high-performance cores can complete tasks more quickly, resulting in power reduction in other areas of the system. Under light CPU load, however, it is more efficient to run tasks on simple, low-power cores.

While migration between cores can be transparent to the OS (as is possible with, e.g., ARM's "big.LITTLE" AMP architecture) a better solution is for the OS to manage a heterogeneous collection of cores itself, powering individual cores on and off reactively.

Alternatively, Intel's Turbo Boost feature, which increases the frequency and voltage of a core when others on the same die are sufficiently idle to keep the chip within its thermal envelope, is arguably a dynamic form of AMP [15].

At the same time, *hotplug* of processors, once the province of specialized machines like the Tandem NonStop systems [6], is becoming more mainstream. More radical proposals for reconfiguring physical processors include Core Fusion [26], whereby multiple independent cores can be morphed into a larger CPU, pooling caches and functional units to improve the performance of sequential programs.

Ultimately, the age of "dark silicon" [21] may well lead to increased core counts, but with a hard limit on the number that may be powered on at any given time. Performance advances and energy savings subsequently will have to derive from specialized hardware for particular workloads or operations [47].

The implications for a future OS are that it must manage a dynamic set of physical cores, and be able to adjust to changes in the number, configuration, and microarchitecture of cores available at runtime, while maintaining a stable execution environment for applications.

2.2 Software

Alongside hardware trends, there is increasing interest in modifying, upgrading, patching, or replacing OS kernels at runtime. Baumann et al. [9] implement dynamic kernel updates in K42, leveraging the object-oriented design of the OS, and later extend this to interface changes using object adapters and lazy update [7]. More recently, Ksplice [3] allows binary patching of Linux kernels without reboot, and works by comparing generated object code and replacing entire functions. Dynamic instrumentation

[1] Barrelfish uses the term *CPU driver* to refer to the kernel-mode code running on a core. In this paper, we use the term "kernel" instead, to avoid confusion with *boot driver*.

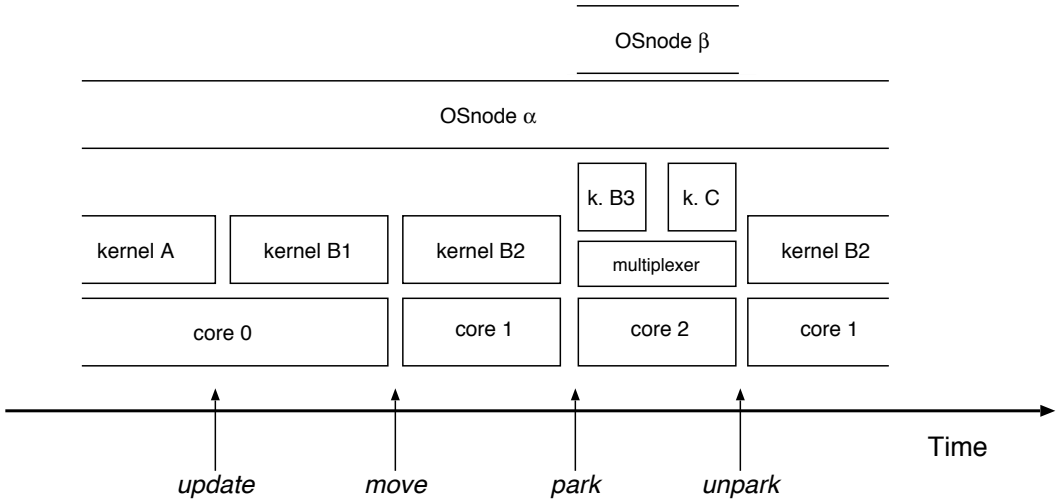


Figure 1: Shows the supported operations of a decoupled OS. **Update**: The entire kernel, dispatching OSnode α, is replaced at runtime. **Move**: OSnode α containing all per-core state, entailing applications is migrated to another core and kernel. **Park**: OSnode α is moved to a new core and kernel that temporarily dispatches two OSnodes. **Unpark**: OSnode α is transferred back to its previous core.

systems like Dtrace [13] provide mechanisms that modify the kernel at run-time to analyze program behavior.

All these systems show that the key challenges in updating an OS online are to maintain critical invariants across the update and to do so with minimal interruption of service (the system should pause, if at all, for a minimal period). This is particularly hard in a multiprocessor kernel with shared state.

In this paper, we argue for addressing *all* these challenges in a single framework for core and kernel management in the OS, although the structure of Unix-like operating systems presents a barrier to such a unified framework. The rest of this paper describes the unified approach we adopted in Barrelfish/DC.

3 Related work

Our work combines several directions in OS design and implementation: core hotplugging, kernel update and replacement, and multikernel architectures.

3.1 CPU Hotplug

Most modern OS designs today support some form of core hotplug. Since the overriding motivation is reliability, unplugging or plugging a core is considered a rare event and the OS optimizes the common case where the cores are not being hotplugged. For example, Linux CPU hotplug uses the `__stop_machine()` kernel call, which halts application execution on all online CPUs for typically hundreds of milliseconds [23], overhead that increases further when the system is under CPU load [25]. We show further evidence of this cost in Section 5.1 where we compare Linux' CPU hotplug with Barrelfish/DC' core update operations.

Recognizing that processors will be configured much more frequently in the future for reasons of energy usage and performance optimization, Chameleon [37] identifies several bottlenecks in the existing Linux implementation due to global locks, and argues that current OSes are ill equipped for processor sets that can be reconfigured at runtime. Chameleon extends Linux to provide support for changing the set of processors efficiently at runtime, and a scheduling framework for exploiting this new functionality. Chameleon can perform processor reconfiguration up to 100,000 times faster than Linux 2.6.

Barrelfish/DC is inspired in part by this work, but adopts a very different approach. Where Chameleon targets a single, monolithic shared kernel, Barrelfish/DC adopts a multikernel model and uses the ability to reboot individual kernels one by one to support CPU reconfiguration.

The abstractions provided are accordingly different: Chameleon abstracts hardware processors behind *processor proxies* and *execution objects*, in part to handle the problem of per-core state (primarily interrupt handlers) on an offline or de-configured processor. In contrast, Barrelfish/DC abstracts the per-core state (typically much larger in a shared-nothing multikernel than in a shared-memory monolithic kernel) behind OSnode and *kernel control block* abstractions.

In a very different approach, Kozuch et al. [30] show how commodity OS hibernation and hotplug facilities can be used to migrate a complete OS between different machines (with different hardware configurations) without virtualization.

Hypervisors are typically capable of simulating hotplugging of CPUs within a virtual machine. Barrelfish/DC can be deployed as a guest OS to manage a variable set of virtual CPUs allocated by the hypervisor. Indeed, Barrelfish/DC addresses a long-standing issue in virtualization: it is hard to fully virtualize the *microarchitecture* of a processor when VMs might migrate between asymmetric cores or between physical machines with different processors. As a guest, Barrelfish/DC can natively handle such heterogeneity and change without disrupting operation.

3.2 Kernel updates

The problem of patching system software without downtime of critical services has been a research area for some time. For example, K42 explored update of a running kernel [7, 9], exploiting the system's heavily object-oriented design. Most modern mainstream OSes support dynamic loading and unloading of kernel modules, which can be used to update or specialize limited parts of the OS.

KSplice [3] patches running Linux kernels without the need for reboot by replacing code in the kernel at a granularity of complete functions. It uses the Linux `stop_machine()` call to ensure that no CPU is currently executing a function to be replaced, and places a branch instruction at the start of the obsolete function to direct execution of the replacement code. Systems like KSplice replace individual functions across all cores at the same time. In contrast, Barrelfish/DC replaces entire kernels, but on a subset of cores at a time. KSplice makes sense for an OS where all cores must execute in the same, shared-memory kernel and the overhead incurred by quiescing the entire machine is unavoidable.

Proteos [22] uses a similar approach to Barrelfish/DC by replacing applications in their entirety instead of applying patches to existing code. In contrast to Ksplice, Proteos automatically applies state updates while preserving pointer integrity in many cases, which eases the burden on programmers to write complicated state transformation functions. In contrast to Barrelfish/DC, Proteos does not upgrade kernel-mode code but focuses on updates for OS processes running in user-space, in a micro-kernel environment. Much of the OS functionality in Barrelfish/DC resides in user-space as well, and Proteos would be applicable here.

Otherworld [18] also enables kernel updates without disrupting applications, with a focus on recovering system crashes. Otherworld can microreboot the system kernel after a critical error without clobbering running applications' state, and then attempt to restore applications that were running at the time of a crash by recreating application memory spaces, open files and other resources.

Rather than relying on a single, system-wide kernel, Barrelfish/DC exploits the multikernel environment to offer both greater flexibility and better performance: kernels and cores can be updated dynamically with (as we show in Section 5) negligible disruption to the rest of the OS.

While their goals of security and availability differ somewhat from Barrelfish/DC, KeyKOS [24] and EROS [42] use partitioned capabilities to provide an essentially stateless kernel. Memory in KeyKOS is persistent, and it allows updates of the OS while running, achieving continuity by restoring from disk-based checkpoints of the entire capability state. Barrelfish/DC by contrast achieves continuity by distributing the capability system, only restarting some of the kernels at a time, and preserving each kernel's portion of the capability system across the restart.

3.3 Multikernels

Multikernels such as fos [48], Akaros [40], Tessellation [33], Hive [14], and Barrelfish [8], are based on the observation that modern hardware is a networked system and so it is advantageous to model the OS as a distributed system. For example, Barrelfish runs a small kernel on each core in the system, and the OS is built as a set of cooperating processes, each running on one of these kernels, sharing no memory, and communicating via message passing. Multikernels are motivated by both the scalability advantages of sharing no cache lines between cores, and the goal of supporting future hardware with heterogeneous processors and little or no cache-coherent or shared physical memory.

Barrelfish/DC exploits the multikernel design for a new reason: dynamic and flexible management of the cores and the kernels of the system. A multikernel can naturally run different versions of kernels on different cores. These versions can be tailored to the hardware, or specialized for different workloads.

Furthermore, since (unlike in monolithic kernels) the state on each core is relatively decoupled from the rest of the system, multikernels are a good match for systems where cores come and go, and intuitively should support reconfiguration of part of the hardware without undue disruption to software running elsewhere on the machine. Finally, the shared-nothing multikernel architecture allows us to wrap kernel state and move it between different kernels without worrying about potentially harmful concurrent accesses.

We chose to base Barrelfish/DC on Barrelfish, as it is readily available, is under active development, supports multiple hardware platforms, and can run a variety of

common applications such as databases and web servers. The features of Barrelfish/DC described in this paper will be incorporated into a future Barrelfish release.

Recently, multikernels have been combined with traditional OS designs such as Linux [27, 36] so as to run multiple Linux kernels on different cores of the same machine using different partitions of physical memory, in order to provide performance isolation between applications. Popcorn Linux [38, 43] boots a modified Linux kernel in this fashion, and supports kernel- and user-space communication channels between kernels [41], and process migration between kernels. In principle, Popcorn extended with the ideas in Barrelfish/DC could be combined with Chameleon in a two-level approach to dynamic processor support.

4 Design

We now describe how Barrelfish/DC decouples cores, kernels, and the rest of the OS. We focus entirely on mechanism in this paper, and so do not address scheduling and policies for kernel replacement, core power management, or application migration. Note also that our main motivation in Barrelfish/DC is adapting the OS for performance and flexibility, and so we do not consider fault tolerance and isolation for now.

We first describe how Barrelfish/DC boots a new core, and then present in stages the problem of per-core state when removing a core, introducing the Barrelfish/DC capability system and kernel control block. We then discuss the challenges of time and interrupts, and finish with a discussion of the wider implications of the design.

4.1 Booting a new core

Current CPU hotplug approaches assume a single, shared kernel and a homogeneous (albeit NUMA) machine, with a variable number of active cores up to a fixed limit, and so a static in-kernel table of cores (whether active or inactive) suffices to represent the current hardware state. Bringing a core online is a question of turning it on, updating this table, and creating per-core state when needed. Previous versions of Barrelfish also adopted this approach, and booted all cores during system initialization, though there has been experimental work on dynamic booting of heterogeneous cores [35].

Barrelfish/DC targets a broader hardware landscape, with complex machines comprising potentially heterogeneous cores. Furthermore, since Barrelfish/DC runs a different kernel instance on each core, there is no reason why the same kernel code should run everywhere – indeed, we show one advantage of *not* doing this in Section 5.3. We thus need an OS representation of a core on the machine which abstracts the hardware-dependent mechanisms for bringing that core up (with some kernel) and down.

Therefore, Barrelfish/DC introduces the concept of a *boot driver*, which is a piece of code running on a "home core" which manages a "target core" and encapsulates the hardware functionality to boot, suspend, resume, and power-down the latter. Currently boot drivers run as processes, but closely resemble device drivers and could equally run as software objects within another process.

A new core is brought online as follows:

1. The new core is detected by some platform-specific mechanism (e.g., ACPI) and its appearance registered with the device management subsystem.
2. Barrelfish/DC selects and starts an appropriate boot driver for the new core.
3. Barrelfish/DC selects a kernel binary and arguments for the new core, and directs the boot driver to boot the kernel on the core.
4. The boot driver loads and relocates the kernel, and executes the hardware protocol to start the new core.
5. The new kernel initializes and uses existing Barrelfish protocols for integrating into the running OS.

The boot driver abstraction treats CPU cores much like peripheral devices, and allows us to reuse the OS's existing device and hotplug management infrastructure [50] to handle new cores and select drivers and kernels for them. It also separates the hardware-specific *mechanism* for booting a core from the *policy* question of what kernel binary to boot the core with.

Boot drivers remove most of the core boot process from the kernel: in Barrelfish/DC we have entirely replaced the existing multiprocessor booting code for multiple architectures (which was spread throughout the system) with boot drivers, resulting in a much simpler system structure, and reduced code in the kernels themselves.

Booting a core (and, indeed, shutting it down) in Barrelfish/DC only involves two processes: the boot driver on the home core, and the kernel on the target core. For this reason, we require no global locks or other synchronization in the system, and the performance of these operations is not impacted by load on other cores. We demonstrate these benefits experimentally in Section 5.1.

Since a boot driver for a core requires (as with a device driver) at least one existing core to execute, there is a potential dependency problem as cores come and go. For the PC platform we focus on here, this is straightforward since any core can run a boot driver for any other core, but we note that in general the problem is the same as that of allocating device drivers to cores.

Boot drivers provide a convenient abstraction of hardware and are also used to shutdown cores, but this is *not* the main challenge in removing a core from the system.

4.2 Per-core state

Taking a core out of service in a modern OS is a more involved process than booting it, since modern multicore OSes include varying amounts of per-core kernel state. If they did not, removing a core would be simply require migrating any running thread somewhere else, updating the scheduler, and halting the core.

The challenge is best understood by drawing a distinction between the *global* state in an OS kernel (i.e., the state which is shared between all running cores in the system) and the *per-core* state, which is only accessed by a single core. The kernel state of any OS is composed of these two categories.

In, for example, older versions of Unix, all kernel state was global and protected by locks. In practice, however, a modern OS keeps per-core state for scalability of scheduling, memory allocation, virtual memory, etc. Per-core data structures reduce write sharing of cache lines, which in turn reduces interconnect traffic and cache miss rate due to coherency misses.

For example, Linux and Windows use per-core scheduling queues, and distributed memory allocators. Corey [10] allowed configurable sharing of page tables between cores, and many Linux scaling enhancements (e.g., [11]) have been of this form. K42 [2] adopted reduced sharing as a central design principle, and introduced the abstraction of *clustered objects*, essentially global proxies for pervasive per-core state.

Multikernels like Barrelfish [8] push this idea to its logical conclusion, sharing no data (other than message channels) between cores. Multikernels are an extreme point in the design space, but are useful for precisely this reason: they highlight the problem of consistent per-core state in modern hardware. As core counts increase, we can expect the percentage of OS state that is distributed in more conventional OSes to increase.

Shutting down a core therefore entails disposing of this state without losing information or violating system-wide consistency invariants. This may impose significant overhead. For example, Chameleon [37] devotes considerable effort to ensuring that per-core interrupt handling state is consistent across CPU reconfiguration. As more state becomes distributed, this overhead will increase.

Worse, how to dispose of this state depends on what it is: removing a per-core scheduling queue means migrating threads to other cores, whereas removing a per-core memory allocator requires merging its memory pool with another allocator elsewhere.

Rather than implementing a succession of piecemeal solutions to this problem, in Barrelfish/DC we adopt a radical approach of lifting *all* the per-core OS state out of the kernel, so that it can be reclaimed lazily without delaying the rest of the OS. This design provides the means to completely decouple per-core state from both the underlying kernel implementation and the core hardware.

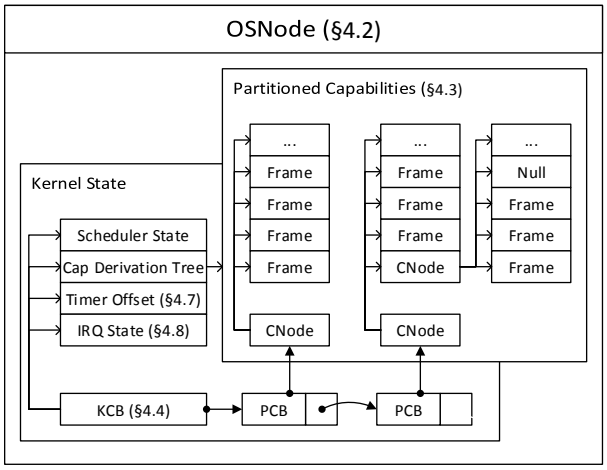


Figure 2: State in the Barrelfish/DC OSnode

We find it helpful to use the term *OSnode* to denote the total state of an OS kernel local to a particular core. In Linux the OSnode changes with different versions of the kernel; Chameleon identifies this state by manual annotation of the kernel source code. In Barrelfish, the OSnode is all the state – there is no shared global data.

4.3 Capabilities in Barrelfish/DC

Barrelfish/DC captures the OSnode using its capability system: all memory and other resources maintained by the core (including interrupts and communication end-points) are represented by capabilities, and thus the OSnode is represented by the capability set of the core. The per-core state of Barrelfish/DC is shown schematically in Figure 2.

Barrelfish/DC's capability system, an extension of that in Barrelfish [44], is derived from the *partitioned capability* scheme used in seL4 [19, 20, 28].

In seL4 (and Barrelfish), all regions of memory are referred to by capabilities, and capabilities are *typed* to reflect what the memory is used for. For example, a "frame" capability refers to memory that the holder can map into their address space, while a "c-node" capability refers to memory that is used to store the bit representations of capabilities themselves. The security of the system as a whole derives from the fact that only a small, trusted computing base (the kernel) holds both a frame capability and a c-node capability to the same memory, and can therefore fabricate capabilities.

A capability for a region can be split into two smaller regions, and also *retyped* according to a set of system rules that preserve integrity. Initially, memory regions are of type "untyped", and must be explicitly retyped to

"frame", "c-node", or some other type.

This approach has the useful property that a process can allocate memory without being able to access its contents. This is used in seL4 to remove *any* dynamic memory allocation from the kernel, greatly simplifying both the formal specification of the kernel and its subsequent proof [20]. All kernel objects (such as process control blocks, or page tables) are allocated by user-level processes which can, themselves, not access them directly.

A key insight of Barrelfish/DC is that this approach can externalize the kernel state *entirely,* as follows.

4.4 Kernel Control Blocks

In developing Barrelfish/DC, we examined the Barrelfish kernel to identify all the data structures which were not direct (optimized) derivations of information already held in the capability tree (and which could therefore be reconstructed dynamically from the tree). We then eliminated from this set any state that did not need to persist across a kernel restart.

For example, the runnable state and other scheduling parameters of a process[2] are held in the process' control block, which is part of the capability system. However, the scheduler queues themselves do not need to persist across a change of kernel, since (a) any scheduler will need to recalculate them based on the current time, and (b) the new scheduler may have a completely different policy and associated data structures anyway.

What remained was remarkably small: it consists of:

- The minimal scheduling state: the head of a linked list of a list of process control blocks.
- Interrupt state. We discuss interrupts in Section 4.8.
- The root of the capability derivation tree, from which all the per-core capabilities can be reached.
- The timer offset, discussed in Section 4.7.

In Barrelfish/DC, we introduce a new memory object, the *Kernel Control Block* (KCB), and associated capability type, holding this data in a standard format. The KCB is small: for 64-bit x86 it is about 28 KiB in size, almost all of which is used by communication endpoints for interrupts.

4.5 Replacing a kernel

The KCB effectively decouples the per-core OS state from the kernel. This allows Barrelfish/DC to shut down a kernel on a core (under the control of the boot driver running on another core) and replace it with a new one. The currently running kernel saves a small amount of persistent state in the KCB, and halts the core. The boot driver then loads a new kernel with an argument supplying the address of the KCB. It then restarts the core (using an IPI on x86 machines), causing the new kernel to boot. This new kernel then initializes any internal data structures it needs from the KCB and the OSnode capability database.

The described technique allows for arbitrary updates of kernel-mode code. By design, the kernel does not access state in the OSnode concurrently. Therefore, having a quiescent state in the OSnode before we shut-down a core is always guaranteed. The simplest case for updates requires no changes in any data structures reachable by the KCB and can be performed as described by simply replacing the kernel code. Updates that require a transformation of the data structures may require a one-time adaption function to execute during initialization, whose overhead depends on the complexity of the function and the size of the OSnode. The worst-case scenario is one that requires additional memory, since the kernel by design delegates dynamic memory allocation to userspace.

As we show in Section 5, replacing a kernel can be done with little performance impact on processes running on the core, even device drivers.

4.6 Kernel sharing and core shutdown

As we mentioned above, taking a core completely out of service involves not simply shutting down the kernel, but also disposing of or migrating all the per-core state on the core, and this can take time. Like Chameleon, Barrelfish/DC addresses this problem by deferring it: we immediately take the core down, but keep the OSnode running in order to be able to dismantle it lazily. To facilitate this, we created a new kernel which is capable of multiplexing several KCBs (using a simple extension to the existing scheduler).

Performance of two active OSnodes sharing a core is strictly best-effort, and is not intended to be used for any case where application performance matters. Rather, it provides a way for an OSnode to be taken out of service in the background, after the core has been shut down.

Note that there is no need for all cores in Barrelfish/DC to run this multiplexing kernel, or, indeed, for any cores to run it when it is not being used – it can simply replace an existing kernel on demand. In practice, we find that there is no performance loss when running a single KCB above a multiplexing kernel.

Decoupling kernel state allows attaching and detaching KCBs from a running kernel. The entry point for kernel code takes a KCB as an argument. When a new kernel is started, a fresh KCB is provided to the kernel code. To restart a kernel, the KCB is detached from the running kernel code, the core is shut down, and the KCB is provided to the newly booted kernel code.

[2]Technically, it is a Barrelfish "dispatcher", the core-local representation of a process. A process usually consists of a set of distinct "dispatchers", one in each OSnode.

We rely on shared physical memory when moving OSnodes between cores. This goes against the original multikernel premise that assumes no shared memory between cores. However, an OSnode is still always in use by strictly one core at the time. Therefore, the benefits of avoiding concurrent access in OSnode state remain. We discuss support for distributed memory hardware in Section 6.

The combination of state externalization via the KCB and kernel sharing on a single core has a number of further applications, which we describe in Section 4.10.

4.7 Dealing with time

One of the complicating factors in starting the OSnode with a new kernel is the passage of time. Each kernel maintains a per-core internal clock (based on a free-running timer, such as the local APIC), and expects this to increase monotonically. The clock is used for per-core scheduling and other time-sensitive tasks, and is also available to application threads running on the core via a system call.

Unfortunately, the hardware timers used are rarely synchronized between cores. Some hardware (for example, modern PCs) define these timers to run at the same rate on every core (regardless of power management), but they may still be offset from each other. On other hardware platforms, these clocks may simply run at different rates between cores.

In Barrelfish/DC we address this problem with two fields in the KCB. The first holds a constant offset from the local hardware clock; the OS applies this offset whenever the current time value is read.

The second field is set to the current local time when the kernel is shut down. When a new kernel starts with an existing KCB, the offset field is reinitialized to the difference between this old time value and the current hardware clock, ensuring that local time for the OSnode proceeds monotonically.

4.8 Dealing with interrupts

Interrupts pose an additional challenge when moving an OSnode between cores. It is important that interrupts from hardware devices are always routed to the correct kernel. In Barrelfish interrupts are then mapped to messages delivered to processes running on the target core. Some interrupts (such as those from network cards) should "follow" the OSnode to its new core, whereas others should not. We identify three categories of interrupt.

1. Interrupts which are used exclusively by the kernel, for example a local timer interrupt used to implement preemptive scheduling. Handling these interrupts is internal to the kernel, and their sources are typically per-core hardware devices like APICs or performance counters. In this case, there is no need to take additional actions when reassigning KCBs between cores.

2. Inter-processor interrupts (IPIs), typically used for asynchronous communication between cores. Barrelfish/DC uses an indirection table that maps OSnode identifiers to the physical core running the corresponding kernel. When one kernel sends an IPI to another, it uses this table to obtain the hardware destination address for the interrupt. When detaching a KCB from a core, its entry is updated to indicate that its kernel is unavailable. Similarly, attaching a KCB to a core, updates the location to the new core identifier.

3. Device interrupts, which should be forwarded to a specific core (e.g. via IOAPICs and PCIe bridges) running the handler for the device's driver.

When Barrelfish/DC device drivers start up they request forwarding of device interrupts by providing two capability arguments to their local kernel: an opaque interrupt descriptor (which conveys authorization to receive the interrupt) and a message binding. The interrupt descriptor contains all the architecture-specific information about the interrupt source needed to route the interrupt to the right core. The kernel associates the message binding with the architectural interrupt and subsequently forwards interrupts to the message channel.

For the device and the driver to continue normal operation, the interrupt needs to be re-routed to the new core, and a new mapping is set up for the (existing) driver process. This could be done either transparently by the kernel, or explicitly by the device driver.

We choose the latter approach to simplify the kernel. When a Barrelfish/DC kernel shuts down, it disables all interrupts. When a new kernel subsequently resumes an OSnode, it sends a message (via a scheduler upcall) to every process which had an interrupt registered. Each driver process responds to this message by re-registering its interrupt, and then checking with the device directly to see if any events have been missed in the meantime (ensuring any race condition is benign). In Section 5.2.1 we show the overhead of this process.

4.9 Application support

From the perspective of applications which are oblivious to the allocation of physical cores (and which deal solely with threads), the additional functionality of Barrelfish/DC is completely transparent. However, many applications such as language runtimes and database systems deal directly with physical cores, and tailor their scheduling of user-level threads accordingly.

For these applications, Barrelfish/DC can use the existing scheduler activation [1] mechanism for process dispatch in Barrelfish to notify userspace of changes in the number of online processors, much as it can already convey the allocation of physical cores to applications.

4.10 Discussion

From a broad perspective, the combination of boot drivers and replaceable kernels is a radically different view of how an OS should manage processors on a machine. Modern general-purpose kernels such as Linux try to support a broad set of requirements by implementing different behaviors based on build-time and run-time configuration. Barrelfish/DC offers an alternative: instead of building complicated kernels that try to do many things, build simple kernels that do one thing well. While Linux selects a single kernel at boot time for all cores, Barrelfish/DC allows selecting not only per-core kernels, but changing this selection on-the-fly.

There are many applications for specialized kernels, including those tailored for running databases or language run-times, debugging or profiling, or directly executing verified user code as in Google's native client [49].

To take one example, in this paper we demonstrate support for hard real-time applications. Despite years of development of real-time support features in Linux and other general-purpose kernels [16], many users resort to specialized real-time OSes, or modified versions of general-purpose OSes [32].

Barrelfish/DC can offer hard real-time support by rebooting a core with a specialized kernel, which, to eliminate OS jitter, has no scheduler (since it targets a single application) and takes no interrupts. If a core is not preallocated, it must be made available at run-time by migrating the resident OSnode to another core that runs a multi-KCB kernel, an operation we call *parking*. If required, cache interference from other cores can also be mitigated by migrating *their* OSnodes to other packages. Once the hard real-time application finishes, the OSnodes can be moved back to the now-available cores. We evaluate this approach in Section 5.3.

5 Evaluation

We present here a performance evaluation of Barrelfish/DC. First (Section 5.1), we measure the performance of starting and stopping cores in Barrelfish/DC and in Linux. Second (Section 5.2), we investigate the behavior of applications when we restart kernels, and when we park OSnodes. Finally, (Section 5.3), we demonstrate isolating performance via a specialized kernel. We perform experiments on the set of x86 machines shown in Table 1. Hyperthreading, TurboBoost, and SpeedStep technologies are disabled in machines that support them, as they complicate cycle counter measurements. TurboBoost and SpeedStep can change the processor frequency in unpredictable ways, leading to high fluctuation for repeated experiments. The same is true for Hyperthreading due to sharing of hardware logic between logical cores. However, TurboBoost and Hyperthreading are both relevant for this work as discussed in Section 6 and Section 1.

packages×cores/uarch	CPU model
2×2 Santa-Rosa	2.8 GHz Opteron 2200
4×4 Shanghai	2.5 GHz Opteron 8380
2×10 SandyBridge	2.5 GHz Xeon E5-2670 v2
1×4 Haswell	3.4 GHz Xeon E3-1245 v3

Table 1: Systems we use in our evaluation. The first column describes the topology of the machine (total number of packages and cores per package) and the second the CPU model.

5.1 Core management operations

In this section, we evaluate the performance of managing cores in Barrelfish/DC, and also in Linux using the CPU Hotplug facility [4]. We consider two operations: shutting down a core (*down*) and bringing it back up again (*up*).

Bringing up a core in Linux is different from bringing up a core in Barrelfish/DC. In Barrelfish/DC, each core executes a different kernel which needs to be loaded by the boot driver, while in Linux all cores share the same code. Furthermore, because cores share state in Linux, core management operations require global synchronization, resulting in stopping application execution in all cores for an extended period of time [23]. Stopping cores is also different between Linux and Barrelfish/DC. In Linux, applications executed in the halting core need to be migrated to other online cores before the shutdown can proceed, while in Barrelfish/DC we typically would move a complete OSnode after the shutdown and not individual applications.

In Barrelfish/DC, the *down* time is the time it takes the boot driver to send an appropriate IPI to the core to be halted plus the propagation time of the IPI and the cost of the IPI handler in the receiving core. For the *up* operation we take two measurements: the boot driver cost to prepare a new kernel up until (and including) the point where it sends an IPI to the starting core (*driver*), and the cost in the booted core from the point it wakes up until the kernel is fully online (*core*).

In Linux, we measure the latency of starting or stopping a core using the log entry of the `smpboot` module and a sentinel line echoed to `/dev/kmsg`. For core shutdown, `smboot` reports when the core becomes offline, and we insert the sentinel right before the operation is initiated.

	Barrelfish/DC						Linux			
	idle			load			idle		load	
	down (µs)	up driver (ms)	up core (ms)	down (µs)	up driver (ms)	up core (ms)	down (ms)	up (ms)	down (ms)	up (ms)
2×2 Santa-Rosa	2.7 / —[a]	29	1.2	2.7 / —	34±17	1.2	131±25	20±1	5049±2052	26±5
4×4 Shanghai	2.3 / 2.6	24	1.0	2.3 / 2.7	46±76	1.0	104±50	18±3	3268± 980	18±3
2×10 SandyBridge	3.5 / 3.7	10	0.8	3.6 / 3.7	23±52	0.8	62±46	21±7	2265±1656	23±5
1×4 Haswell	0.8 / —[a]	7	0.5	0.8 / —	7±0.1	0.5	46±40	14±1	2543±1710	20±5
	Results in cycles									
	$\times 10^3$	$\times 10^6$	$\times 10^6$	$\times 10^3$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$
2×2 Santa-Rosa	8 / —	85	3.4	8 / —	97±49	3.5	367± 41	56±2.0	14139±5700	74±21
4×4 Shanghai	6 / 6	63	2.6	6 / 7	115±192	2.6	261±127	44±2.0	8170±2452	46± 8
2×10 SandyBridge	9 / 10	27	2.1	9 / 10	59±133	2.1	155±116	53±2.0	5663±4141	57±12
1×4 Haswell	3 / —	26	1.9	2.9 / —	26±0.40	2.0	156±137	50±0.5	8647±5816	69±16

Table 2: Performance of core management operations for Barrelfish/DC and Linux (3.13) when the system is idle and when the system is under load. For the Barrelfish/DC *down* column, the value after the slash shows the cost of stopping a core on another socket with regard to the boot driver. [a]We do not include this number for Santa-Rosa because it lacks synchronized timestamp counters, nor for Haswell because it only includes a single package.

For core boot, `smpboot` reports when the operation starts, so we insert the sentinel line right after the operation.

For both Barrelfish/DC and Linux we consider two cases: an idle system (*idle*), and a system with all cores under load (*load*). In Linux, we use the `stress` tool [45] to spawn a number of workers equal to the number of cores that continuously execute the `sync` system call. In Barrelfish/DC, since the file-system is implemented as a user-space service, we spawn an application that continuously performs memory management system calls on each core of the system.

Table 2 summarizes our results. We show both time (msecs and µsecs) and cycle counter units for convenience. All results are obtained by repeating the experiment 20 times, and calculating the mean value. We include the standard deviation where it is non-negligible.

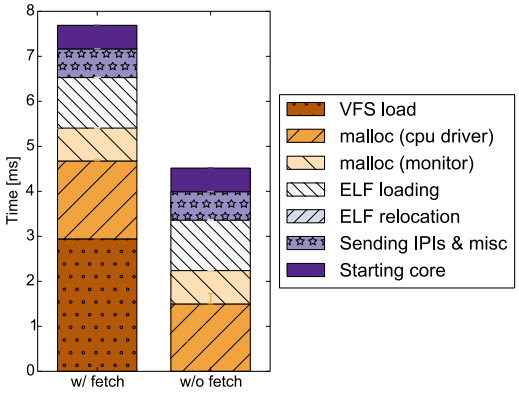


Figure 3: Breakdown of the cost of bringing up a core for the Haswell machine.

Stopping cores: The cost of stopping cores in Barrelfish/DC ranges from 0.8 µs (Haswell) to 3.5 µs (SandyBridge). Barrelfish/DC does not share state across cores, and as a result no synchronization between cores is needed to shut one down. Furthermore, Barrelfish/DC' shutdown operation consists of sending an IPI, which will cause the core to stop after a minimal operation in the KCB (saving the timer offset). In fact, the cost of stopping a core in Barrelfish/DC is small enough to observe the increased cost of sending an IPI across sockets, leading to an increase of 5% in stopping time on SandyBridge and 11% on Shanghai. These numbers are shown in Table 2, in the Barrelfish/DC *down* columns after the slash. As these measurements rely on timestamp counters being synchronized across packages, we are unable to present the cost increase of a cross-socket IPI on the Santa-Rosa machine whose timestamp counters are only synchronized within a single package.

In stark contrast, the cost of shutting down a core in Linux ranges from 46 ms to 131 ms. More importantly, the shutdown cost in Linux explodes when applying load, while it generally remains the same for Barrelfish/DC. For example, the average time to power down a core in Linux on Haswell is increased by 55 times when we apply load.

Starting cores: For Barrelfish/DC, the setup cost in the boot driver (*driver*) dominates the cost of starting a core (*core*). Fig. 3 shows a breakdown of the costs for bringing up a core on Haswell. *Starting core* corresponds to the

core Table 2 column, while the rest corresponds to operations performed by the boot driver: loading the image from storage, allocating memory, ELF loading and relocation, etc. Loading the kernel from the file system is the most expensive operation. If multiple cores are booted with the same kernel, this image can be cached, significantly improving the time to start a core as shown in the second bar in Fig. 3. We note that the same costs will dominate the restart operation since shutting down a core has negligible cost compared to bringing it up. Downtime can be minimized by first doing the necessary preparations in the boot driver and then halting and starting the core.

Even though Barrelfish/DC has to prepare the kernel image, when idle, the cost of bringing up a core for Barrelfish/DC is similar to the Linux cost (Barrelfish/DC is faster on our Intel machines, while the opposite is true for our AMD machines). Bringing a core up can take from 7 ms (Barrelfish/DC/Haswell) to 29 ms (Barrelfish/DC/Santa-Rosa). Load affects the cost of booting up a core to varying degrees. In Linux such an effect is not observed in the Shanghai machine, while in the Haswell machine load increases average start time by 33%. The effect of load when starting cores is generally stronger in Barrelfish/DC (e.g., in SandyBridge the cost is more than doubled) because the boot driver time-shares its core with the load generator.

Overall, Barrelfish/DC has minimal overhead stopping cores. For starting cores, results vary significantly across different machines but the cost of bringing up cores in Barrelfish/DC is comparable to the respective Linux cost.

5.2 Applications

In this section, we evaluate the behavior of real applications under two core management operations: *restarting*, where we update the core kernel as the application runs, and *parking*. In parking, we run the application in a core with a normal kernel and then move its OSnode into a multi-KCB kernel running on a different core. While the application is parked it will share the core with another OSnode. We use a naive multi-KCB kernel that runs each KCB for 20 ms, which is two times the scheduler time slice. Finally, we move the application back to its original core. The application starts by running alone on its core. We execute all experiments in the Haswell machine.

5.2.1 Ethernet driver

Our first application is a Barrelfish NIC driver for the Intel 82574, which we modify for Barrelfish/DC to re-register its interrupts when instructed by the kernel (see Section 4.8). During the experiment we use `ping` from a client machine to send ICMP echo requests to the NIC. We run `ping` as root with the `-A` switch, where the inter-packet intervals adapt to the round-trip time. The `ping` manual states: "on networks with low rtt this mode is essentially equivalent to flood mode."

Fig. 4a shows the effect of the restart operation in the round-trip time latency experienced by the client. Initially, the ping latency is 0.042 ms on average with small variation. Restarting the kernel produces two outliers (packets 2307 and 2308 with an RTT of 11.1 ms and 1.07 ms, respectively). Note that 6.9 ms is the measured latency to bring up a core on this machine (Table 2).

We present latency results for the parking experiment in a timeline (Fig. 4b), and in a cumulative distribution function (CDF) graph (Fig. 4c). Measurements taken when the driver's OSnode runs exclusively on a core are denoted *Exclusive*, while measurements where the OSnode shares the core are denoted *Shared*. When parking begins, we observe an initial latency spike (from 0.042 ms to 73.4 ms). The spike is caused by the parking operation, which involves sending a KCB capability reference from the boot driver to the multi-KCB kernel as a message.[3] After the initial coordination, outliers are only caused by KCB time-sharing (maximum: 20 ms, mean: 5.57 ms). After unparking the driver, latency returns to its initial levels. Unparking does not cause the same spike as parking because we do not use messages: we halt the multi-KCB kernel and directly pass the KCB reference to a newly booted kernel.

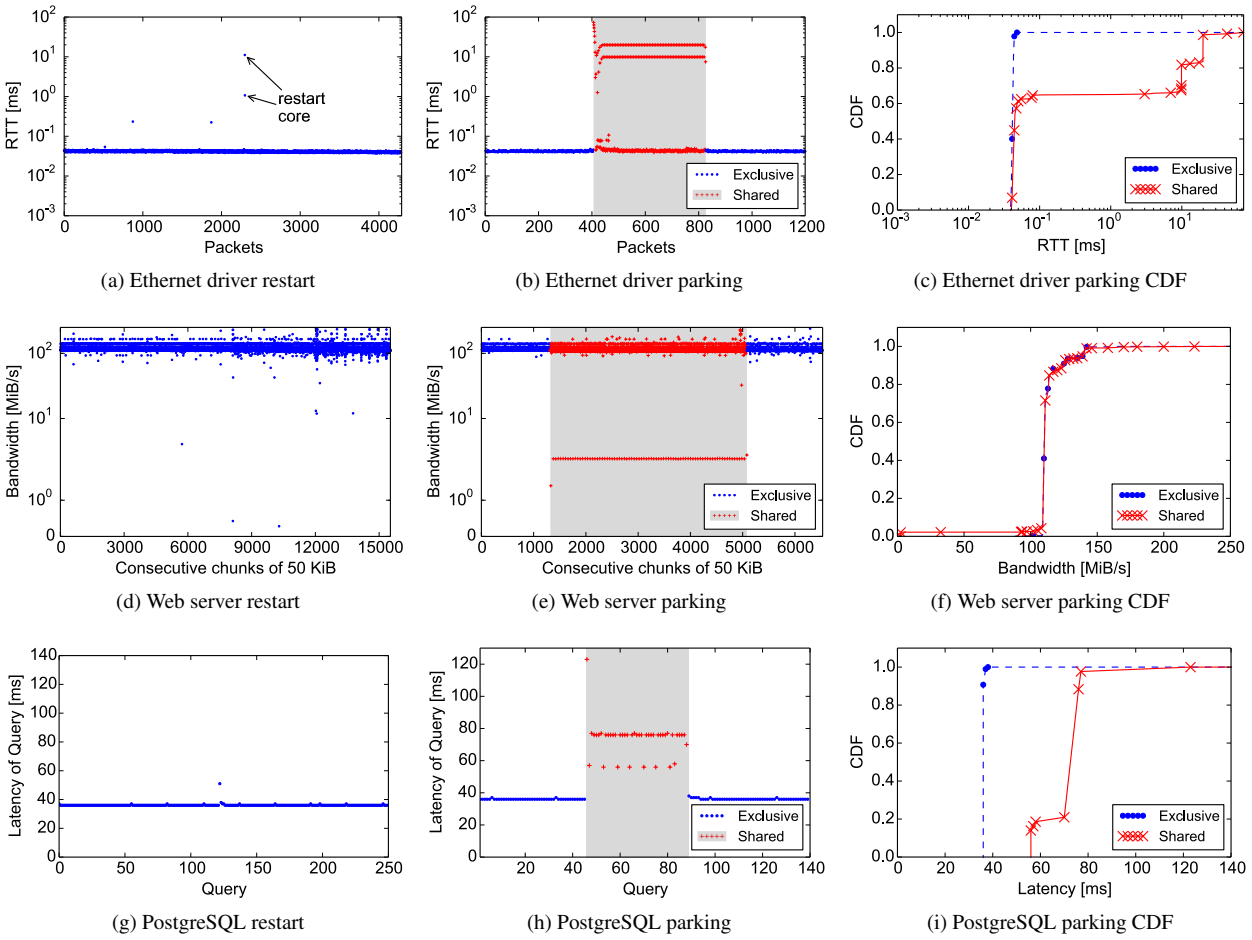
5.2.2 Web server

In this experiment we examine how a web server[4] that serves files over the network behaves when its core is restarted and when its OSnode is parked. We initiate a transfer on a client machine in the server's LAN using `wget` and plot the achieved bandwidth for each 50 KiB chunk when fetching a 1 GiB file.

Fig. 4d shows the results for the kernel restart experiment. The effect in this case is negligible on the client side. We were unable to pinpoint the exact location of the update taking place from the data measured on the client and the actual download times during kernel updates were indistinguishable from a normal download. As expected, parking leads to a number of outliers caused by KCB time-sharing (Figures 4e and 4f). The average bandwidth before the parking is 113 MiB/s and the standard deviation 9 MiB/s, whereas during parking the average bandwidth is slightly lower at 111 MiB/s with a higher standard deviation of 19 MiB/s.

[3] We follow the Barrelfish approach, where kernel messages are handled by the *monitor*, a trusted OS component that runs in user-space.

[4] The Barrelfish native web server.



(a) Ethernet driver restart (b) Ethernet driver parking (c) Ethernet driver parking CDF

(d) Web server restart (e) Web server parking (f) Web server parking CDF

(g) PostgreSQL restart (h) PostgreSQL parking (i) PostgreSQL parking CDF

Figure 4: Application behavior when restarting kernels and parking OSnodes. For each application we include a timeline graph for restarting, and a timeline and a CDF graph for parking.

5.2.3 PostgreSQL

Next, we run a PostgreSQL [39] database server in Barrelfish/DC, using TPC-H [46] data with a scaling factor of 0.01, stored in an in-memory file-system. We measure the latency of a repeated CPU-bound query (query 9 in TPC-H) on a client over a LAN.

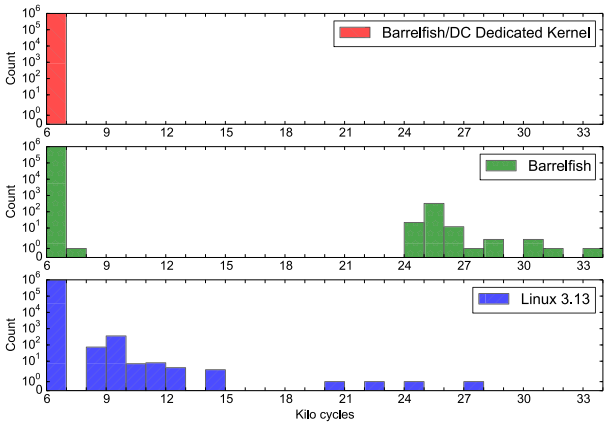
Fig. 4g shows how restart affects client latency. Before rebooting, average query latency is 36 ms. When a restart is performed, the first query has a latency of 51 ms. After a few perturbed queries, latency returns to its initial value.

Figures 4h and 4i show the effect of parking the OSnode that contains the PostgreSQL server. As before, during normal operation the average latency is 36 ms. When the kernel is parked we observe two sets of outliers: one (with more points) with a latency of about 76 ms, and one with latency close to 56 ms. This happens, because depending on the latency, some queries wait for two KCB time slices (20 ms each) of the co-hosted kernel, while others wait only for one.

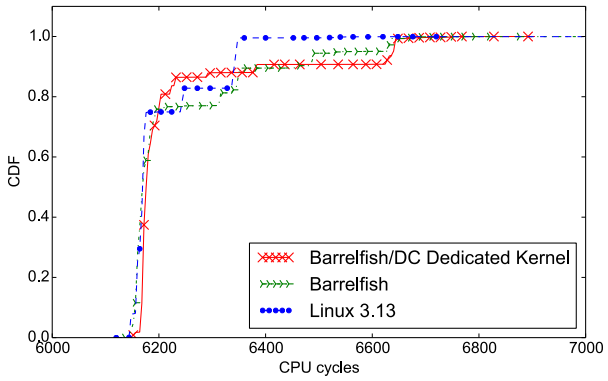
Overall, we argue that kernel restart incurs acceptable overhead for online use. Parking, as expected, causes a performance degradation, especially for latency-critical applications. This is, however, inherent in any form of resource time-sharing. Furthermore, with improved KCB-scheduling algorithms the performance degradation can be reduced or tuned (e.g., via KCB priorities).

5.3 Performance isolation

Finally, we illustrate the benefits of Barrelfish/DC' support for restarting cores with specialized kernels using the case of hard-real time applications where eliminating OS jitter is required. To ensure that the application will run uninterrupted, we assign a core with a specialized kernel that does not implement scheduling and does not handle interrupts (see Section 4.10). We evaluate the performance isolation that can be achieved with our



(a) Histogram for all samples



(b) CDF for samples in the range of 6–7k cycles

Figure 5: Number of cycles measured for 10^3 iterations of a synthetic benchmark for Barrelfish/DC, Barrelfish, and Linux using real-time priorities.

specialized kernel compared to the isolation provided by: (i) an unmodified Barrelfish kernel, and (ii) a Linux 3.13 kernel where we set the application to run with real-time priority. We run our experiments on the Haswell machine, ensuring that no other applications run on the same core.

To measure OS jitter we use a synthetic benchmark that only performs memory stores to a single location. Our benchmark is intentionally simple to minimize performance variance caused by architectural effects. We sample the timestamp counter every 10^3 iterations, for a total of 10^6 samples. Fig. 5a shows a histogram of sampled cycles, where for all systems, most of the values fall into the 6-7 thousand range (i.e., 6-7 cycles latency per iteration). Fig. 5b presents the CDF graph for the 6–7 kcycles range, showing that there are no significant differences for the three systems in this range.

Contrarily to the Barrelfish/DC dedicated kernel where *all* of the samples are in the 6-7k range, in Linux and Barrelfish we observe significant outliers that fall outside this range. Since we run the experiment on the same hardware, under the same configuration, we attribute the outliers to OS jitter. In Barrelfish the outliers reach up to 68k cycles (excluded from the graph). Linux performs better than Barrelfish, but its outliers still reach 27–28 kcycles. We ascribe the worse behavior of Barrelfish compared to Linux to OS services running in user-space.

We conclude that Barrelfish/DC enables the online deployment of a dedicated, simple to build, OS kernel that eliminates OS jitter and provides hard real-time guarantees.

6 Future directions

Our ongoing work on Barrelfish/DC includes both exploring the broader applications of the ideas, and also removing some of the existing limitations of the system.

On current hardware, we plan to investigate the power-management opportunities afforded by the ability to replace cores and migrate the running OS around the hardware. One short-term opportunity is to fully exploit Intel's Turbo Boost feature to accelerate a serial task by temporarily vacating (and thereby cooling) neighboring cores on the same package.

We also intend to use core replacement as a means to improve OS instrumentation and tracing facilities, by dynamically instrumenting kernels running on particular cores at runtime, removing all instrumentation overhead in the common case. Ultimately, as kernel developers we would like to minimize whole-system reboots as much as possible by replacing single kernels on the fly.

Barrelfish/DC currently assumes cache-coherent cores, where the OS state (i.e., the OSnode) can be easily migrated between cores by passing physical addresses. The lack of cache-coherency per se can be handled with suitable cache flushes, but on hardware platforms without shared memory, or with different physical address spaces on different cores, the OSnode might not require considerable transformation to migrate between cores. The Barrelfish/DC capability system *does* contain all the information necessary to correctly swizzle pointers when copying the OSnode between nodes, but the copy is likely to be expensive, and dealing with shared-memory application state (which Barrelfish fully supports outside the OS) is a significant challenge.

A somewhat simpler case to deal with is moving an OSnode between a virtual and physical machine, allowing the OS to switch from running natively to running in a VM container.

Note that there is no requirement for the boot driver to share memory with its target core, as long as it has a mechanism for loading a kernel binary into the latter's address space and controlling the core itself.

When replacing kernels, Barrelfish/DC assumes that the OSnode format (in particular, the capability system)

remains unchanged. If the in-memory format of the capability database changes, then the new kernel must perform a one-time format conversion when it boots. It is unclear how much of a limitation this is in practice, since the capability system of Barrelfish has changed relatively little since its inception, but one way to mitigate the burden of writing such a conversion function is to exploit the fact that the format is already specified in a domain-specific high-level language called Hamlet [17] to derive the conversion function automatically.

While Barrelfish/DC decouples cores, kernels, and the OS state, the topic of appropriate *policies* for using these mechanisms without user intervention is an important area for future work. We plan to investigate policies that, based on system feedback, create new kernels to replace others, and move OSnodes across cores.

Finally, while Barrelfish/DC applications are notified when the core set they are running on changes (via the scheduler activations mechanism), they are currently insulated from knowledge about hardware core reconfigurations. However, there is no reason why this must always be the case. There may be applications (such as databases, or language runtimes) which can benefit from being notified about such changes to the running system, and we see no reason to hide this information from applications which can exploit it.

7 Conclusion

Barrelfish/DC presents a radically different vision of how cores are exploited by an OS and the applications running above it, and implements it in a viable software stack: the notion that OS state, kernel code, and execution units should be decoupled and freely interchangeable. Barrelfish/DC is an OS whose design assumes that all cores are dynamic.

As hardware becomes more dynamic, and scalability concerns increase the need to partition or replicate state across cores, system software will have to change its assumptions about the underlying platform, and adapt to a new world with constantly shifting hardware. Barrelfish/DC offers one approach to meeting this challenge.

8 Acknowledgements

We would like to thank the anonymous reviews and our shepherd, Geoff Voelker, for their encouragement and helpful suggestions. We would also like to acknowledge the work of the rest of the Barrelfish team at ETH Zurich without which Barrelfish/DC would not be possible.

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems 10*, 1 (1992), 53–79.

[2] APPAVOO, J., DA SILVA, D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems 25*, 3 (2007).

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the EuroSys Conference* (2009), pp. 187–198.

[4] ASHOK RAJ. CPU hotplug support in the Linux kernel. `https://www.kernel.org/doc/Documentation/cpu-hotplug.txt`.

[5] The Barrelfish Operating System. `http://www.barrelfish.org/`, 12.04.14.

[6] BARTLETT, J. F. A NonStop Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles* (1981), pp. 22–29.

[7] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the USENIX Annual Technical Conference* (2007), pp. 1–14.

[8] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009), pp. 29–44.

[9] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference* (2005), pp. 279–291.

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (2008), pp. 43–57.

[11] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), pp. 1–8.

[12] BUTLER, M., BARNES, L., SARMA, D. D., AND GELINAS, B. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro 31*, 2 (Mar. 2011), 6–15.

[13] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference* (2004), pp. 15–28.

[14] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 12–25.

[15] CHARLES, J., JASSI, P., S, A. N., SADAT, A., AND FEDOROVA, A. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2009).

[16] CORBET, J. Deadline scheduling for 3.14. `http://www.linuxfoundation.org/news-media/blogs/browse/2014/01/deadline-scheduling-314`, 12.04.14.

[17] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the 5th Workshop on Programming Languages and Operating Systems* (2009).

[18] DEPOUTOVITCH, A., AND STUMM, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the EuroSys Conference* (2010), pp. 181–194.

[19] DERRIN, P., ELKADUWE, D., AND ELPHINSTONE, K. *seL4 Reference Manual*. NICTA, 2006. `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.

[20] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems* (2008), pp. 35–40.

[21] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011), pp. 365–376.

[22] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and automatic live update for operating systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 279–292.

[23] GLEIXNER, T., MCKENNEY, P. E., AND GUITTOT, V. Cleaning up Linux's CPU hotplug for real time and energy management. *SIGBED Rev. 9*, 4 (Nov. 2012), 49–52.

[24] HARDY, N. KeyKOS Architecture. *SIGOPS Operating Systems Review 19*, 4 (1985), 8–25.

[25] CPU hotplug. `https://wiki.linaro.org/WorkingGroups/PowerManagement/Archives/Hotplug`, 12.04.14.

[26] IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. F. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), pp. 186–197.

[27] JOSHI, A. Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system. In *Proceedings of the Linux Symposium* (2010), pp. 101–108.

[28] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009).

[29] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro 25*, 2 (2005), 21–29.

[30] KOZUCH, M. A., KAMINSKY, M., AND RYAN, M. P. Migration without virtualization. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems* (2009), pp. 10–15.

[31] KUMAR, R., FARKAS, K. I., JOUPPI, N. P., RANGANATHAN, P., AND TULLSEN, D. M. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), pp. 81–92.

[32] Real-time Linux. `https://rt.wiki.kernel.org/`, 12.04.14.

[33] LIU, R., KLUES, K., BIRD, S., HOFMEYR, S., ASANOVIĆ, K., AND KUBIATOWICZ, J. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism* (2009).

[34] MARR, D. T., DESKTOP, F. B., HILL, D. L., HINTON, G., KOUFATY, D. A., MILLER, J. A., AND UPTON, M. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* (Feb 2002).

[35] MENZI, D. Support for heterogeneous cores for Barrelfish. Master's thesis, Department of Computer Science, ETH Zurich, July 2011.

[36] NOMURA, Y., SENZAKI, R., NAKAHARA, D., USHIO, H., KATAOKA, T., AND TANIGUCHI, H. Mint: Booting multiple Linux kernels on a multicore processor. In *Proceedings of the International Conference on Broadband and Wireless Computing, Communication and Applications* (2011), pp. 555–560.

[37] PANNEERSELVAM, S., AND SWIFT, M. M. Chameleon: Operating system support for dynamic processors. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), pp. 99–110.

[38] Popcorn Linux. `http://popcornlinux.org/`, 12.04.14.

[39] PostgreSQL. `http://www.postgresql.org/`, 12.04.14.

[40] RHODEN, B., KLUES, K., ZHU, D., AND BREWER, E. Improving per-node efficiency in the datacenter with new OS abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), pp. 25:1–25:8.

[41] SADINI, M., BARBALACE, A., RAVINDRAN, B., AND QUAGLIA, F. A Page Coherency Protocol for Popcorn Replicated-kernel Operating System. In *Proceedings of the ManyCore Architecture Research Community Symposium (MARC)* (Oct. 2013).

[42] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: A Fast Capability System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 170–185.

[43] SHELDON, B. H. Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system. Master's thesis, Virginia Polytechnic Institute and State University, May 2013.

[44] SINGHANIA, A., KUZ, I., AND NEVILL, M. Capability Management in Barrelfish. Technical Note 013, Barrelfish Project, ETH Zurich, December 2013.

[45] Stress Load Generator. `http://people.seas.harvard.edu/~apw/stress/`, 12.04.14.

[46] TPC-H. `http://www.tpc.org/tpch/`, 12.04.14.

[47] VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. Conservation Cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010), pp. 205–218.

[48] WENTZLAFF, D., GRUENWALD III, C., BECKMANN, N., MODZELEWSKI, K., BELAY, A., YOUSEFF, L., MILLER, J., AND AGARWAL, A. An operating system for multicore and clouds: Mechanisms and implementation. In *ACM Symposium on Cloud Computing (SOCC)* (June 2010).

[49] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), pp. 79–93.

[50] ZELLWEGER, G., SCHUEPBACH, A., AND ROSCOE, T. Unifying Synchronization and Events in a Multicore OS. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems* (2012).