

# ROS: A Scalable Operating System for Parallel Applications on Many-core Architectures<sup>\*</sup>

Barret Rhoden, Kevin Klues, Andrew Waterman, David Zhu, Eric Brewer

*Computer Science Division, University of California at Berkeley*

{brho, klueska, waterman, yuzhu, brewer}@eecs.berkeley.edu

## Abstract

The emergence of many-core architectures provides a rare chance to redesign operating systems, including the interfaces they expose to an application. We introduce a new operating system, called ROS, designed specifically to address the limitations of current OSs as we move into the many-core era. Our goals are (1) to provide better support for parallel applications for both high-performance and general-purpose computing, and (2) to scale the kernel to thousands of cores. ROS is based on three related ideas: structure the OS asymmetrically, provision guaranteed resources, and treat parallel processes as a single native entity, which we call a *many-core process* (MCP). We have a working prototype of ROS that runs on x86 and SPARC V8.

In this paper, we describe the design and current implementation of the central features of ROS and provide a preliminary evaluation of its ability to meet its goals. We show that ROS's resource guarantees protect performance sensitive parallel applications from OS noise and competing applications, especially as the number of cores increases. Our initial results validate the basic tenants of ROS and prove its value as a platform for research on many-core operating systems.

## 1 Introduction

Current operating systems were originally designed for uniprocessor systems. These systems have had SMP support for years, but it was initially added with a small number of nodes in mind. Today, Linux supports up to 4096 nodes in theory [21], but not in practice [10], and the system is an evolution from the original SMP design. The kernel itself has been modified to scale past previous bottlenecks, but there has been no fundamental change

for the many-core era. For example, every node potentially runs all of the kernel code and participates equally in the management of resources and scheduling. Furthermore, modern kernels are not designed to explicitly support parallel processes.

Parallel applications are performance sensitive to the underlying state of the machine and to any OS processing that occurs [30, 15]. It is essential for the application to be able to gather information about the state of the system and to make requests that influence the decisions made by the OS [1, 2]. Additionally, the system must contain mechanisms to ensure performance isolation between competing applications.

Many of these insights come from the HPC community, which is appropriate due to the similarities between many-core chips and shared-memory supercomputers. However, while HPC workloads are parallel, they do not represent the needs of general-purpose computing. Many-core consumer devices should enable interactive applications with highly parallel components, such as speech recognition or live video processing [24].

In light of these goals, we created a new many-core OS, called ROS, designed from the ground up to provide better support for a wide range of parallel applications and improved kernel scalability. We present the design and implementation of ROS, specifically focusing on a new process abstraction we call the *many-core process* (MCP) and discussing how we guarantee resources allocated to this abstraction. Briefly, an MCP is a single OS task with one address space that runs gang-scheduled on multiple cores (rather than a set of processes, one per core). The OS grants resources to MCPs on demand, but also partitions resources as a means of provisioning (as opposed to allocation), giving the system more freedom to utilize these resources when not in use.

We provide a preliminary evaluation that shows ROS's ability to provide predictable, isolated performance and low, guaranteed response times for parallel applications. A prototype of ROS already runs parallel applications

<sup>\*</sup>Research supported by Microsoft Award #024263 and Intel Award #024894 and by matching funding from U.C. Discovery (Award #DIG07-102270).

that link against GNU `libc` and our own custom pthread library on a 32-core Intel X7560 Nehalem-EX and on a 64-core FPGA-based SPARC hardware emulator [32]. Our results show that even at low levels of parallelism, OS noise and overhead can be amplified by sensitive parallel applications. We also see that ROS’s MCP abstraction is on the right track to providing a predictable, isolated environment for the execution of parallel programs in the face of competing applications and changing resource allocations.

## 2 Related Work

A number of operating systems have recently emerged that also target future many-core architectures. Corey [10] is an exokernel that improves on kernel scalability by reducing sharing to the bare minimum desired by an application. Corey’s notion of kernel cores are similar to the Coarse-Grained cores we grant to MCPs, which we discuss shortly. The Multikernel [6] focuses on scalability in a NUMA and possibly heterogeneous environment, where they scale by distributing state and running message passing algorithms to achieve consensus. Helios [28] focuses on seamlessly running on heterogeneous hardware. We do not explicitly focus on NUMA or heterogeneity; our work is orthogonal to these concerns. None of these systems share our focus of directly supporting parallel applications or provide an abstraction like the MCP.

Our MCPs are rooted in older systems such as the Exokernel [12] and scheduler activations [1]. Like the Exokernel, ROS exposes information about the underlying system to the application writers to allow them to make the best decisions. Like scheduler activations, the ROS kernel does not interfere with the scheduling of threads in parallel processes. Neither of these share our joint focus on kernel scalability and resource provisioning.

Like ROS, traditional systems such as Linux can provide some performance isolation. Through pinning and routing interrupts to specific cores, one can cut down on the noise and jitter caused by the OS. However, certain tasks still interfere with applications and hurt their predictability and performance. For instance, each core on Linux has a timer interrupt and a variety of per-core kernel worker threads. These threads include the migration thread, `ksoftirqd`, the generic event workqueue, `kblockd` and others. In general, they preempt the application and perform housekeeping or other managerial work, inducing jitter and noise. We believe that ROS will provide better performance isolation, and show some preliminary results in Section 5.

ROS has been heavily influenced by the HPC community, where issues of parallel application performance have been explored for decades. Although we are not tar-

geting massive clusters, many-core chips will encounter similar issues to shared-memory supercomputers, and we can learn from their lessons. For instance, several recent studies [30, 15, 19] have analyzed the interference caused by the operating system. Certain parallel applications, especially those with barriers and other communication collectives, can magnify the overhead of the OS due to the unpredictable nature of preemptions and background tasks. Low-frequency, long-duration interruptions from background tasks are especially damaging [15]. ROS is designed to provide a noise-free environment for the processing of parallel applications, similar to many of the light-weight kernels in the HPC community.

Unlike HPC systems, ROS supports consumer-oriented workloads such as those commonly seen on a workstation or mobile phone. Processes are interactive, with latency requirements and deadlines, and the machine is not often dedicated to one specific task. To this end, ROS supports multiplexing multiple parallel applications on the machine, including preempting resources for processes that have provisioned resources.

Our model for resource provisioning is grounded in a large body of prior work, mostly generated by research on multi-core operating systems and the real-time community [16, 26, 27, 37]. In particular, our model provides an abstraction similar to that of a *Software Performance Unit* (SPU) [34], with a few distinctions. SPUs rely on processes to actively share unused resources with background tasks, instead of allowing the system to manage these resources. We do not grant a process access to its provisioned resources, until the process explicitly requests them. Instead, provisioned resources are similar to the notion of a *reserve* [25]. However, *reserves* are static and are designed for applications that have a pre-determined schedule.

One might consider using virtual machines (VMs) for resource isolation. Although very useful, VMs are simply the wrong abstraction for *interacting* applications on a single system. They needlessly couple protection with resource management, do not normally allow flexible sharing or reallocation of resources, and do not allow the interaction among processes in separate VMs that is typical on a consumer system. Our view that protection domains should be decoupled from resource management is similar to resource containers [4], which allow accurate accounting of resource utilization, independent of the execution context. Our resource provisioning scheme is complementary to their approach.

## 3 Design

ROS is designed for kernel scalability and native support for parallel applications. It is based on three re-

lated ideas: structure the OS asymmetrically, provision guaranteed resources, and treat parallel processes as a single native entity, the *many-core process*, which gang-schedules one address space across many cores.

### 3.1 Asymmetric Use of Cores

There are two aspects to the asymmetry of ROS: the use of cores and the execution of OS services.

In order to increase both scalability and performance, most cores are gang-scheduled as a group using MCPs at a relatively coarse granularity, currently 50ms. The groups imply both fewer management decisions and less OS state per core, since most of the state is per MCP, not per core. The coarse granularity implies less overhead and better cache performance due to fewer context switches. We call such cores *coarse-grained cores* (CG). The gang-scheduling also enables efficient spin-based concurrency control, as it is very likely that the lock-holder is actively running. In general, CG cores are left alone by the kernel, are isolated from other cores, and perform their own scheduling. All of these help to provide MCPs with a predictable execution environment.

The asymmetric alternative to CG cores is *low-latency cores* (LL), which are not gang-scheduled and execute with a short quantum, currently 1ms, and thus finer granularity. LL cores are used for traditional single-core processes (SCPs), kernel housekeeping tasks, interrupt handling, and low-latency handlers for networking or UI processing. The work of the OS itself is also done primarily on LL cores. By handling the low-latency requirements, LL cores decouple the CG quantum from the response time of the system, and thus allow the CG cores to be more coarsely scheduled and thus more efficient.

The actual scheduling *decisions* for CG cores, covered in more detail below, are made on LL cores. This is possible because the MCP that runs on a set of CG cores is just a single entity to the OS and because the scheduling within an MCP is handled by the MCP itself. In addition, there is no need for timer interrupts on CG cores, just on the LL cores that make the decisions. Once the decision is made, ROS provides the mechanism to quickly switch a set of CG cores to a new MCP. In contrast, with traditional processes, the set of involved cores work independently with their own timers and then must achieve consensus to enforce gang scheduling. In general in ROS, relatively few cores make nearly all of the decisions.

The second kind of asymmetry involves system calls. ROS allows threads on CG cores to invoke syscalls on LL cores, to enable concurrency between the application and the OS, especially for I/O. We call these *Asynchronous Remote Calls* (ARCs). For heavyweight syscalls, such as I/O commands, ARCs lead to better cache locality and less lock contention. In particular, the CG core

avoids cache pollution, and the LL core (that executes the syscall) avoids some locking (as more state is private to that core) and also has better cache locality.

However, other syscalls, such as setting a local timer, are either inherently local or too short, and should not be shipped to another core. The policies around which syscalls should be offloaded is the subject of future work and will not be addressed in this paper.

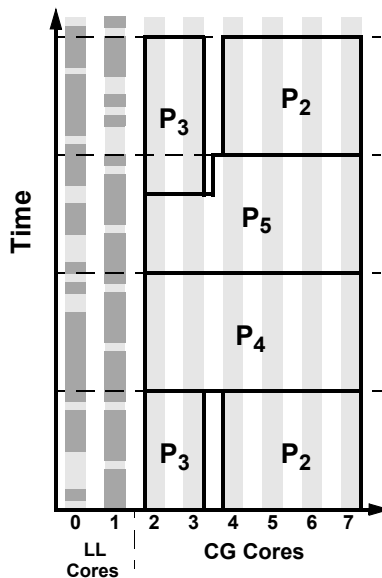


Figure 1: An instance of ROS running on 8 cores with time-slicing of MCPs, SCPs, interrupt handlers, and kernel tasks. MCPs are scheduled on the CG cores generally on the virtual boundaries shown by the dashed lines. The various handlers, kernel tasks and traditional single-core processes execute on the LL cores. The “notch” in  $P_5$  occurs because  $P_3$  needed its provisioned cores back, and thus two of  $P_5$ ’s cores were preempted by ROS and given to  $P_3$ .

Figure 1 shows the asymmetric use of 8 cores over time. The two cores on the left are LL cores, with many small tasks, while the rightmost six cores are CG cores with few context switches and large, relatively stable allocations. It is the large allocations, both in time and space, that lead to scalability and efficiency. The dashed horizontal lines demark the coarse-grain boundaries, but they are virtual boundaries on CG cores as there are no timer interrupts. Instead, only the leftmost core has a timer interrupt on these boundaries, at which point it decides whether or not to change the allocation, and if so executes the change by interrupting the relevant CG cores.

Although most of the context switches for CG cores occur on these virtual boundaries, switches may also occur when an MCP yields some or all its cores, or when

the OS decides to preempt a running MCP and replace it with another one. In the figure, this happens when  $P_3$  takes over two of the cores that were in use by  $P_5$ . This most commonly occurs because the cores were provisioned to  $P_3$  and it needs them back quickly, which we cover next.

### 3.2 Provisioning Resources

ROS provisions resources for guaranteed use by a process<sup>1</sup> A resource is defined as anything sharable in the system, including cores, RAM, cache, on- and off-chip memory bandwidth, access to I/O devices, etc. We assume that hardware/software mechanisms exist to control the partitioning, isolation, and QoS of resources. Some of these mechanisms are already available (e.g. cores, RAM, caches via page coloring), while others may only be available in future hardware (e.g. on-chip memory bandwidth). We use the term *partition* when talking about a dedicated chunk of a particular resource.

A provision allows an application to reserve the resources it needs to meet its latency requirements, but still enables the reuse of those resources when they are underutilized. We expect the underutilization of provisioned resources to be common for interactive apps or apps that deal with audio/video, sensor data, or networking. Yielding underutilized resources also provides opportunities for increased energy efficiency [20]. Later we show that a background task can effectively share the underutilized cores of a live video application without hurting its latency.

A provisioned resource is available to a process on demand; the process simply needs to ask and the resource will be allocated. Decoupling provisioning from allocation allows the OS to utilize unused resources throughout the system. The important part of provisioning is that the rightful owner of a resource can acquire the resource quickly, in time to meet a latency deadline. ROS currently supports provisioning of cores, as we show in Section 5.

A resource guarantee means that the OS will ensure the process has exclusive use of its allocated resources. These resources can be provisioned in both space and time, allowing processes to declare their resource needs in both dimensions [22]. For example, a process might indicate that it needs exclusive access to 25 cores every other coarse time slice.

Processes can always acquire more resources than have been provisioned for them, but there is no guarantee that they will be able to retain those resources if the system becomes over-utilized. This flexibility leads to better utilization of system resources and reduces the

<sup>1</sup>Although we currently use processes as the entity to which we allocate resources, any reasonable resource container could be used [4].

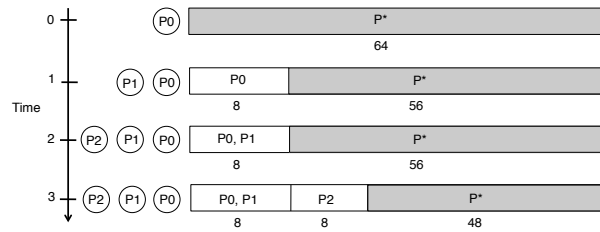


Figure 2: Example provisioning of 64 cores to a set of processes over time. At each time step some number of processes exist in the system (as indicated by the circles on the left), and some number of provisions are set up for a group of processes (as indicated by the bars on the right).  $P^*$  is shorthand for the list of all processes that currently exist in the system.

hard problem of deciding when to revoke a resource from a process to the simpler problem of deciding which processes can provision resources in the first place (i.e. admission control). Policy decisions such as from which process to revoke a resource, or how to discourage applications from hoarding resources are beyond the scope of this paper.

Using our model, resource provisions may be created in one of two ways: (1) programmatically by some process in the process hierarchy, or (2) by a system administrator who wants fine-grained control over how resources are partitioned throughout the system. The OS may also provision resources to a group of processes, allowing for flexible administration policies. By associating multiple processes with each resource provision, we essentially decouple the notion of protection from resource management.

As a simple example, consider Figure 2, which shows how one might spatially partition a set of 64 cores among 3 competing processes. At time 0, there is only one process in the system,  $P_0$ , and a single partition containing all cores. At time 1, a second process,  $P_1$ , has been created and 8 cores have been provisioned for  $P_0$ . At time 2,  $P_1$  has been added to the group of processes associated with the 8 core partition, and a third process  $P_2$  has appeared. Finally, at time 3, 8 more cores have been provisioned for  $P_2$ . Given this example, if all cores were currently allocated to process  $P_2$  at the end of Time 3, and a request for 8 cores came in from process  $P_0$ , those cores would be immediately revoked from  $P_2$  and given to  $P_0$ . No matter how many requests come in from other processes, however,  $P_2$  will still be able to hold onto at least its 8 provisioned cores.

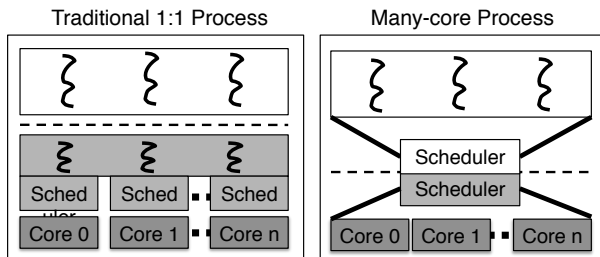


Figure 3: On the left is a traditional 1:1 threading model with each user-space thread mapped onto a kernel thread scheduled by the kernel. On the right is our MCP model with the kernel scheduling physical cores and user-space scheduling the threads that run on them.

### 3.3 Many-core Processes

Processes in ROS share many similarities with traditional processes. They run in the lowest privilege mode, are the unit of protection and control in the kernel, communicate with each other, and are executing instances of a program. We extend this model in the following ways:

- Resources, such as cores and memory, are explicitly granted and revoked. The kernel exposes information about a process’s current resource allocation and the system’s utilization, via shared read-only pages, and allows the process to make requests based on this information.
- Most cores running a process’s address space are gang scheduled on CG cores; scheduling decisions for gangs are made at a coarse granularity.
- There are no kernel tasks or threads underlying each thread of a process, unlike in a 1:1 or M:N threading model. As shown in Figure 3, there is only one kernel object per MCP, regardless of the number of cores involved. The user-level scheduler is typically a linked-in library (e.g. pthreads).
- Traditionally blocking system calls are asynchronous and non-blocking, and may also be offloaded via ARCs.
- A process will not lose a core (or other granted resource) without being warned. Typically, the MCP yields the core shortly after notification, but if not, the core will be preempted.
- Most faults are redirected to and handled by the process, typically via library code provided by ROS. This enables ACPM-style user-level memory management [17], although it is not yet implemented.

#### 3.3.1 Benefits to Kernel Scalability

These changes enable the kernel to scale well as the number of cores increases. Since we do not have a kernel task underlying every user-space context, we can manage the entire parallel process as one entity. We only need one process descriptor, instead of  $n$ . Because we are not scheduling  $n$  independent kernel tasks, we can remove per-core run queues and avoid the overhead of load-balancing them.

The changes to the process abstraction also result in memory savings. The kernel does not maintain a stack for every context of every process, which will be important for parallel processes that request large amounts of cores. The core kernel is event based and uses (essentially) continuations to store per-task state, as previously done in Capriccio [35]. This enables many concurrent kernel events, such as blocking I/Os, without requiring large amounts of memory for stack space.

#### 3.3.2 Benefits to Parallel Applications

ROS provides the notion of a virtual multiprocessor, similar to scheduler activations [1]. A process can request a block of cores, and the OS guarantees that all cores allocated will run simultaneously [14]. We differ from scheduler activations in that there is only one kernel object per process and that we alert the application before any of its cores are actually revoked. Additionally, the OS will not send unexpected interrupts to cores allocated to a process, except when necessary for preemption by higher priority processes. These cores are called virtual cores; when a physical core is allocated, a virtual core is pinned to that physical core.

There are many advantages to this approach for parallel applications. Classic shared-memory synchronization methods, such as spin-locks, depend on gang scheduling for reasonable performance. By not servicing periodic interrupts on gang-scheduled cores, we eliminate jitter in finely tuned applications’ runtimes. Since blocking system calls are asynchronous, I/O concurrency is decoupled from processing concurrency and processes do not need to request extra cores for I/O processing. For memory, processes have knowledge of which virtual pages are resident and page faults are reflected back to the process, allowing it to maintain control over its cores. Finally, the kernel scheduler does not decide which user thread runs at a given time, removing its ability to adversely impact performance through a poor decision. The application knows best which threads are more important than others, such as those in critical sections. A user-level scheduler, such as Lithe [29], can schedule its contexts on the cores granted by the kernel.

### 3.3.3 Event Notification and Preemption

If desired, a process can be notified of events (including an impending preemption) by telling the kernel how it wants to be notified and on which virtual core. Methods of notification include a combination of a message with parameters, a bit in a bitmask, or an interprocessor interrupt (IPI). For instance, a process can request to receive an IPI on a specific virtual core whenever it loses a resource or when the kernel needs to reclaim memory. This IPI triggers an event handler, giving the application an opportunity to yield some of its resources. Certain notifications need to be sent to the virtual core that caused the event, such as a divide by zero exception or a page fault. Processes can also notify their own virtual cores or receive notifications from per-core alarms, which can be used to implement preemptive user-level thread schedulers.

Since our MCPs are part of a larger system with many applications, eventually its cores and other resources will be revoked. The kernel will notify the process (if desired) before it revokes a core or any other resource. This gives the process a chance to save its context or finish a critical section and then yield the core. This does not need to be an IPI notification. For instance, to avoid being preempted in a critical section, low-level locks in a threading library can check the flag for an impending preemption before locking.

Any notification IPIs sent to a virtual core that is not currently mapped to a physical core, which could be due to yielding or preemption, will be handled when that virtual core is restarted. Based on the desired policy, applications that do not respond to preemption warnings can be preempted with no warning in the future to reduce the preemption latency.

### 3.3.4 Short Asynchronous Event Handlers

Not all processes need to run in parallel on their cores all the time. MCPs are gang-scheduled at a coarse granularity; however, applications may want fast response times for short blocks of code. For example, UI events and packet acknowledgements need to execute quickly, and may want to avoid the overhead of context switching in all of a process's cores. A program may want to do some work in response to an event to determine if it wants to bring up the gang of virtual cores. To support this, processes can register handlers that will run independently from the gang, with access to the entire address space. These handlers run on LL cores in response to certain events, but with a short timeslice and possibly when the gang is not scheduled. These handlers decouple bulk processing from interactive responses.

### 3.3.5 Using Many-core Processes

The MCP is more complicated than a traditional process. Its interfaces for resource management, notification, and preemption are intended for expert programmers and library writers. Most applications can link against libraries tuned for a particular platform or workload and benefit from ROS. As an example, we implemented a customized pthread library on top of the MCP interface. Any application (including legacy ones) can link with our pthread library and run as an MCP. Due to the information exposed by the kernel, we can tune these low-level libraries to be more efficient, similar to Exokernel [13]. For instance, `barrier_wait()` does not need to yield or sleep if the code detects that there are enough virtual cores to run all of the threads participating in the barrier. Even simpler applications that do not care about performance and do not link against a smart library still can ignore this information and the kernel will treat them like a traditional single-core process, and run them on an LL core.

## 4 Implementation

ROS is an event-driven operating system that runs on both x86 and SPARC V8. There is one kernel stack per core that is reused for every entry to the kernel, and there are no kernel stacks underlying processes or virtual cores. The kernel uses a slab allocator [9] for dynamic memory allocation, page coloring for cache partitioning, and a networked computer for file system support.

We ported GNU `libc` to ROS, support dynamically and statically linked ELF executables, provide thread-local storage (TLS), and have basic user-level libraries to take advantage of ROS's features. We have ported the pthread library to run on ROS and plan to support other user-level threading libraries such as Intel's TBB [18] and Lithe [29] in the near future. We picked the C library as the point of compatibility instead of a kernel interface, such as the Linux ABI, because it simplifies the kernel and is sufficient for our purposes. Our long-term plan for binary compatibility is to leverage the Palacios project [33] to support virtual machines within MCPs.

### 4.1 Page Coloring

We integrated page coloring [7, 23] into our page allocator, which allows us to partition caches among different applications. Any given physical page maps to a specific set in a cache. All of the pages that map to a given set are said to have the same *color*. Page coloring works by carefully selecting physical page numbers for virtual memory mappings, such that a process's address space

can only index into a limited chunk of the physically indexed cache.

Our implementation determines how many colors are supported in the last level cache and creates lists of free pages associated with each color. We then allocate a range of colors to a process, depending on the desired cache partitioning. All memory allocations for that process will come from the lists for its allocated colors.

There are many limitations of using page coloring for cache partitioning. Partitioning the caches is needlessly coupled with partitioning the available physical pages for virtual memory mappings. Changing a process's cache partition on the fly is expensive since it requires copying page contents. Coloring the kernel's memory is troublesome, especially since it often needs physically contiguous pages. Finally, partitioning one level of cache, such as a shared last level cache, will also partition any lower level caches, which may be undesirable. For instance, when we run an MCP on a dedicated CG core, it will only have access to part of its L1 and L2, even though no other process will run on that core for a while. A thorough evaluation of page coloring for many-core architectures is the subject of future work.

## 4.2 Kernel Messaging

We developed a messaging subsystem in the kernel to facilitate process management. These kernel messages direct cores to execute functions (with arguments) in the order in which the messages were sent, much like Active Messages [36]. Messages are used to start, notify, preempt, and kill a process's virtual cores. Initially, we used IPIs and specific interrupt handlers to execute these functions, but these could not handle various corner cases related to concurrent messages and did not provide arguments. The in-order execution of kernel messages greatly simplified the process management code.

Kernel messages come in two flavors: Immediate and Routine. Immediate messages execute once received, and they must return to the previously-executing context. An example of an immediate message is a TLB shoot-down. Routine messages do not need to return, and they will be executed at points when the kernel has cleaned up its state such that it can leave its stack. These points are when the kernel halts the core due to lack of work or whenever returning to user-space after a syscall. An example of a routine message is the message to start a process on a core. This function will not return, since it jumps to user-space, so the kernel must make sure it finishes its current work, releases any locks, and restores references before executing the message. Neither type of message will be received while interrupts are disabled.

Kernel messages are implemented on x86 and SPARC by allocating a message structure from the slab allocator

and enqueueing it in the destination core's list and sending an IPI. Other architectures may implement them differently, perhaps using Active Messaging hardware [36]. One benefit of using the slab allocator is that sending a message cannot fail, which helps to avoid tricky deadlock scenarios in the process management code.

Currently, kernel messages are point-to-point, though we plan to support broadcast messaging in the future. All process management messages are the same for a given process, so that we can broadcast a message to an entire MCP.

## 4.3 MCP Management

MCPs have a few structures that differ from a traditional process, most of which are stored in two memory regions called *procinfo* and *procddata*. These are shared-memory regions mapped into a process's address space and are used to communicate information to and from user-space asynchronously and without context switches. *Procinfo* is read-only information that the kernel exposes to a process. *Procddata* is read-write data that either the user or the kernel can modify.

The most important structure in *procinfo* is the Virtual Core Map (Vcoremap), which maintains the bidirectional mapping from virtual core to physical core, as well as some other bookkeeping. A process can use this to determine its physical core, which is useful for cache-conscious user-level schedulers. *Procddata* contains structures for the management of preemption and notification, including storage of thread contexts the kernel preempts, and queues for notification events. *Procddata* also contains a ring buffer for ARCs, using Xen's ring buffer code [5].

All processes start as single-core processes (SCPs) and must explicitly transition into many-core mode by requesting a virtual core. When an MCP gets a virtual core, the kernel initializes a fresh context and starts it at the program's entry point (`._start` for an Elf). The kernel sets the stack pointer to an address in the process's address space that was `mmap'd` by the process before transitioning into many-core mode. Each virtual core has one of these stacks, called *transition stacks*, on which it can swap between user-level threads or handle notifications. Virtual cores initialize the thread local storage (TLS) of the transition context and store their virtual core id in the TLS for easy access. Any code can determine its virtual core ID by looking in the TLS and can determine its physical core ID by indexing in the Vcoremap.

Compared to a traditional OS, ROS virtual core management is simpler and requires less memory. When a process asks for a new virtual core, the kernel allocates a physical core, adds it to the Vcoremap, and sends a start message. There is no process structure allocation, struc-

ture initialization, or management. Scheduling decisions still only consider one entity, the MCP, instead of each individual virtual core. There are no kernel stacks either, which saves another page. On a typical Linux system, each “virtual core” requires a 1.7KB task struct and a 4KB stack. ROS virtual cores currently require 976B on x86, the majority of which is storage space for registers, including floating-point and SSE registers.

## 4.4 Notification

Processes find out about system events through a notification mechanism. For each type of notification, a process writes an entry in *procd* expressing if, how, and on which virtual core it would like to be notified. All notifications come as either a structure called a notification event, which carries arguments, or as a bit in a bit-mask. These notification events are written to a per-vcore shared queue in *procd*.

Since *procd* is writable by a potentially malicious process and we wanted to keep all parts of the queue together, we created a data structure we call the Bounded Concurrent Queue (BCQ). BCQs provide a safe, concurrent, non-blocking means for the kernel to deliver information to a process. A BCQ is a fixed size array that can handle multiple producers and consumers where the producers do not trust the consumers and the consumers have access to the producers index pointers. The worst thing a consumer can do is cause the producer to be unable to find a free spot, including the case when the queue is full, or overwrite an unconsumed entry. Multiple producers synchronize by reserving space in the queue by using compare-and-swap (CAS) on the index variable. The producers will attempt a limited number of CASs, aborting the operation if they fail too many times or if the queue is full. Future versions of BCQs will soon use fetch-and-add with a buffer size that is a power of two, similar to Xen’s ring buffers [5]. Such an implementation will improve performance and eliminate the possibility of starvation.

For urgent events, a process can request to be actively notified, in which case the kernel will send a routine kernel message. When the kernel processes the routine message, it will save the integer registers of the previously running context into the virtual core’s spot in *procd*. Then the kernel will start up the virtual core with a fresh set of registers at the program entry point, just like starting the virtual core for the first time. We treat startup and notification similarly since it simplifies both the kernel and the low-level user-space code. When the process has finished handling its notifications, it can restart the context that the kernel saved.

Faults and exceptions are reflected back to the process via the notification mechanism. Page faults are also re-

flected, so a process must not fault while handling a notification or otherwise is in transition context. The kernel will kill processes that fault on their transition stacks to prevent recursion, so processes pin their transition stacks in memory. Since MCPs are already managing which pages are resident and which are evicted for performance reasons, this extra restriction is not overly burdensome.

To avoid unnecessary active notifications and to prevent cascading notifications, there is a per-virtual core flag for whether or not notifications are enabled. This mechanism is analogous to disabling interrupts. The kernel will set this bit upon notification and the process will clear it when it is ready to receive notifications again. Our low-level libraries provide code to leave a transition stack, restart a halted context, and enable notifications, which is analogous to returning from an interrupt.

There is a race between when the process’s virtual core checks for new notifications and when it reenables notifications. The kernel may have wanted to send a new active notification while notifications were disabled. We solve this through a technique similar to a wakeup-waiting switch [31]: the kernel sets a flag for a pending notification after posting notifications to *procd*. The process clears this flag before checking for notifications, empties its queue, and transitions back to its original work. The process checks the notification pending flag, and if it is set it knows it missed an active notification and makes a syscall to self-notify its virtual core.

## 4.5 Preemption

When the kernel wants to preempt a virtual core, it will first warn the process and then later preempt the core. Specifically, a warning is a field in *procinfo* with the time stamp of when the kernel will preempt the core. A process can simply check this value to see that a preemption is pending. The process can also request to be notified for impending preemptions and yield the core to the kernel. This mechanism is in contrast to scheduler activations which must preempt two cores in order to communicate the loss of one core.

If a process is unresponsive, the kernel will send a preemption kernel message to the given core. The kernel then saves the integer registers and the floating point registers. When that virtual core is restarted, the kernel will restart the preempted context instead of a fresh context. If a notification is pending when the virtual core restarts and if notifications are enabled, the kernel loads the floating point state and restarts the virtual core as if it had received an active notification. A notification would be pending if an event happened while the virtual core was offline.

Ideally, processes will yield their cores when warned and notified. One reason they may be delayed is because

they are already processing another notification and have not had a chance to notice the impending preemption. Part of our future work includes determining a reasonable amount of time to wait for a response.

## 5 Evaluation

In this section we evaluate ROS in terms of its ability to isolate processes both from each other and from the kernel, as well as its ability to respond quickly to dynamically changing resource requests. Additionally, we provide a set of microbenchmarks, which characterizes the overhead of performing common ROS operations, such as starting and stopping processes, requesting virtual cores, sending preemption notifications, etc. We perform our evaluation using a set of applications ported to ROS from the PARSEC [8] benchmark suite. Specifically, we use the barrier-based *fluidanimate* application and the *x264* parallel video encoder. Both applications link directly against our MCP-based pthreads library discussed in Section 3, with only minimal changes to the applications themselves (e.g. adding hooks to gather performance data).

For our microbenchmarks, we perform our experiments on two different platforms. The first is a 32-core (64-core SMT), Quad-socket X7560 Nehalem-EX running at 2.27GHz with 64GB of RAM. Although this machine has the ability to run up to 64 distinct hardware threads, we disable SMT support to reduce interference from interleaving. The second is an FPGA-based simulator called RAMP Gold [3], which can model up to 64 in-order 1-GHz SPARC V8 cores in a single shared-memory hierarchy. While inherently slower, having a RAMP port of ROS allows us to quickly try out new hardware features that ROS can benefit from as we move forward. Most notable are the abilities to accurately simulate a large number of cores, as well as partition resources such as memory bandwidth directly on chip.

### 5.1 Microbenchmarks

Tables 1 and 2 summarize the results of our microbenchmarks. Table 1 shows the time it takes to perform common kernel operations as they interact with an MCP. Table 2 shows the time it takes to perform common process operations as they interact with the kernel. All microbenchmarks were run 12 times, and the numbers presented are the means of these runs.

The purpose of these microbenchmarks is to provide a general idea of how long certain operations take. Many of the larger numbers are the subject of future improvements. Getting a cold vcore is the time it takes to create the transition stacks and data structures in the program to handle the new virtual core. This includes the time it

	Nehalem ( $\mu$ s)	RAMP ( $\mu$ s)
Preempt One Vcore	2	2
Warn-Preempt One Vcore	3	1
Preempt Max Vcores	165	18
Warn-Preempt Max Vcores	462	36
Kill an MCP	218	265

Table 1: Microbenchmarks of common ROS kernel operations on both the Quad-socket X7560 Nehalem-EX and RAMP Gold. A “warn-preempt” is when the process yields in response to a preemption notification.

	Nehalem ( $\mu$ s)	RAMP ( $\mu$ s)
Create a new SCP	469	258
Get One Vcore (Warm)	4	2
Get One Vcore (Cold)	15	18
Get Max Vcores (Warm)	40	17
Get Max Vcores (Cold)	3448	1254
Preempt and Get Max Vcores	201	55

Table 2: Microbenchmarks of common Process operations on both the Quad-socket X7560 Nehalem-EX and RAMP Gold.

takes the kernel to clear pages, which is not optimized yet. Once a vcore is “warm”, reallocating it takes little time.

The other major source of latency in these benchmarks is the kernel messaging system. For example, preempting 64 cores requires 64 kernel messages, one to each core. This leads to the large times for getting, preempting, and killing MCPs. Another optimization we will make is to build a broadcast kernel messaging system, leveraging hardware support for broadcast IPIs.

The main takeaway from this section is that it does not take a long time to recover all of a process’s provisioned virtual cores.

### 5.2 Isolating Performance

As mentioned in Section 2, OS jitter and interference from competing applications are two of the primary sources of performance degradation for many HPC-based workloads. Moving forward, the need to limit these types of interference are becoming increasingly important even in the general computing space, especially for multimedia or cloud-based applications, which have deadlines or require some level of predictable performance.

In the following set of experiments, we measure ROS’s ability to isolate different processes both from each other and from the kernel. Both experiments have been run using the barrier-based *fluidanimate* application

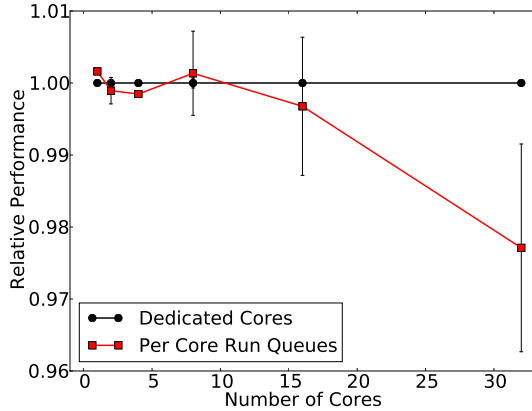


Figure 4: Relative running times of *fluidanimate*, as compared between ROS with and without OS noise.

from the PARSEC benchmark suite. This application was chosen in part because it uses barrier synchronization, which is sensitive to outside interference.

### 5.2.1 OS Jitter

In our first experiment we measure ROS’s ability to isolate processes from OS jitter. As discussed in Section 3, MCPs typically run exclusively on coarse-grained cores, while low-latency cores take care of handling interrupts on behalf of an MCP. For this experiment, we run the ROS kernel on a single low-latency core and *fluidanimate* as an MCP on a varying number of coarse-grained cores: [1,32] on the Nehalem<sup>2</sup>. All threads run on dedicated virtual cores, and all threads synchronize on 8 different barriers throughout the execution of the application.

As a point of comparison, we built a facility into ROS that mimics a 1% overhead due to OS interference, whereby each core receives an interrupt every 100ms and spins for 1ms. Such interference is representative of background OS processing and the load-balancing of per-core run queues. Figure 4 shows the results of this experiment, as run with and without OS noise. As the number of cores increases, when the OS interferes slightly, the performance decreases and the standard deviation increases. As expected for barrier-based parallel applications, a slight amount of interference amplifies poor performance, which MCPs avoid.

<sup>2</sup>*Fluidanimate* only runs with a power of 2 number of threads, limiting the number of data points we could collect for this experiment

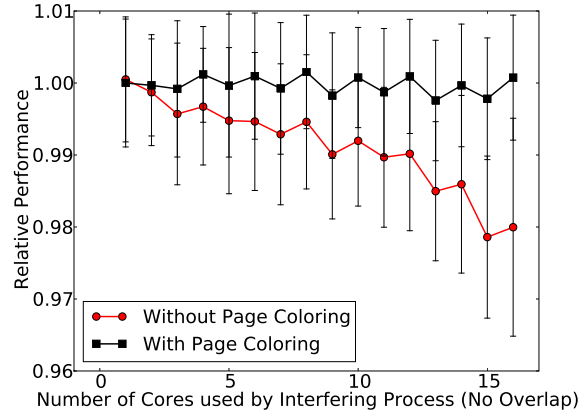


Figure 5: Normalized Running times of *fluidanimate* in the presence of a “cache busting” interferer, with and without page coloring.

### 5.2.2 Application Interference

In our second experiment we measure ROS’s ability to isolate different processes from one another. Typically, application interference can degrade performance in two ways. First, applications that share cores need to be time-multiplexed, thus slowing down both their executions times. Second, applications on different cores can influence one another’s behavior by (inadvertently) trashing a shared cache. Since, by design, there is minimal sharing of cores in ROS, we evaluate its ability to avoid application interference by partitioning all shared caches using standard page coloring techniques.

For this experiment, we run a single instance of *fluidanimate* on half of the available cores, and a second “cache busting” application on a varying subset of the remaining cores: [0,16] on the Nehalem. Independent of how many cores it runs on, the cache busting application always allocates memory equal to twice the size of the last level cache. It is then divided equally among each of the cores and walked through with a stride equal to the size of a cache line (i.e. 64 bytes). Figure 5 summarizes the results of this experiment, as performed once with page coloring enabled, and once with page coloring disabled. As expected, without cache partitioning, the performance of *fluidanimate* decreases as the amount of interference increases. The reason for the variance in the performance is most likely due to contention for memory bandwidth. Future versions of ROS will support memory bandwidth partitioning once it is supported in our hardware platforms.

### 5.3 Responding to Changing Resources

In our third experiment, we evaluate ROS’s ability to respond to dynamically changing resource requests. In Section 3, we introduced the notion of resource provisioning, and explained how MCP’s can take advantage of provisioning to guarantee exclusive access to a set of resources whenever they require them. For this experiment, we provision all cores on the machine to a modified version of the *x264* video encoder from the PARSEC benchmark suite. We modified this application to read data from a dedicated network stream, so as to simulate real-time processing in the presence of deadlines. We then stream a 100 frame video with a resolution of  $1280 \times 1024$  over the network at a rate of 20Fps. *x264* waits for an entire frame to arrive, requests 16 cores, and then starts processing the frame. Once it has completed processing of the frame, it frees all of but one of its cores back to the system, and begins to periodically poll while it waits for the next frame to arrive. Along side *x264*, we run an embarrassingly parallel compute-bound application (i.e. a `while(1)` loop) that is able to take advantage of the cores freed by *x264* while it waits for its next frame. Figure 6 shows how well ROS is able to respond to an incoming resource request from *x264*. When a request comes in, it sends a preempt pending notification to the `while(1)` loop running on all cores. Once the `while(1)` loop receives this notification, it yields all of its cores. ROS then grants these cores to *x264*. This cycle then continues until *x264* has processed the entire streaming data file. As seen in the Figure, even in the presence of a competing application, *x264* is able to process its frames in a comparable amount of time to when it is running alone. Furthermore, the system is able to reallocate *x264*’s idle cores to the `while(1)` loop.

## 6 Future Work

One next step involves the interactions between *low-latency cores* (LL) and *coarse-grained cores* (CG). We plan to explore the tradeoffs of running OS services, such as the filesystem, on LL cores using ARCs. ARCs ought to have performance benefits for cache locality and locking, as discussed in Section 3. However, shipping the work to another core has a higher overhead than trapping locally; for a “null” syscall on x86, an ARC takes about 200–400ns, compared to 100ns for (local) `sysenter`. For short or embarrassingly parallel syscalls, servicing the call locally on CG cores will probably be the best choice. It will be interesting to see at what point it is beneficial to use ARCs and more importantly if we can decide dynamically based on workload, since remote work only helps if there are idle cores.

We also plan to investigate splitting an application’s

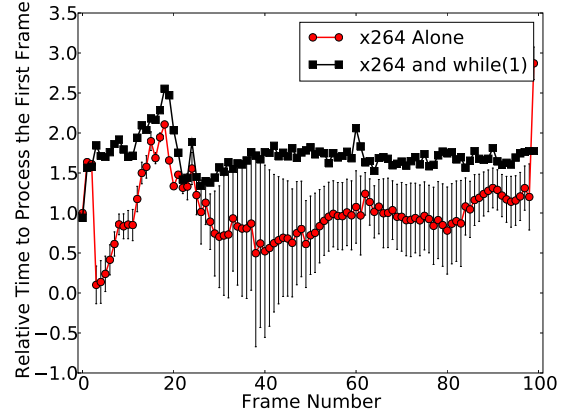


Figure 6: *x264* streaming video encoder with competing application and provisioned cores. The x-axis denotes the frame number processed by *x264*, and the y-axis denotes the relative processing time for that frame.

work between LL cores and CG cores. Specifically, we will implement the short handlers discussed in Section 3.3.4 and use them to process UI events for an application, while the bulk of the computation executes on CG cores. One way to do this is to run the UI as a set of low-latency handlers that invoke parallel libraries on CG cores for decoding, object recognition, etc.

Finally, we will convert the networking subsystem to be aware of the distinction between LL cores and CG cores. TCP ACKs and the control path will run on LL cores, and the data will be shipped directly to the CG cores using fast page remapping similar to Fbufs [11]. The TCP ACKs can be handled by the kernel or even by an application. The short application handlers would allow us to make a low-latency TCP networking stack in user-space.

There are many other avenues that we can explore with ROS. Since the kernel is relatively small and well understood, we can run a variety of interesting experiments. We also work closely with the developers of RAMP Gold and can explore how small hardware changes allow more effective OS capabilities. One example of this is providing hardware support for cache partitioning and comparing it to the effectiveness of the software-only approach of page coloring.

## 7 Conclusion

We presented a new many-core operating system called ROS, designed to natively support parallel applications and scale to large numbers of cores. ROS borrows from both the HPC community and the traditional systems community; it leverages the lessons of HPC while re-

maintaining responsive to the needs of future general purpose computing. ROS runs real, mostly unmodified applications on both x86 and RAMP Gold, which will provide us with many interesting possibilities in hardware/software co-design.

We described the *many-core process* abstraction (MCP), discussed its benefits, and showed its potential to properly support performance sensitive parallel applications through a preliminary evaluation.

Finally, we showed we could provision cores to a latency-sensitive application, have that application yield its cores when it blocked on I/O, and reacquire its cores in time to meet the deadline for the next computational phase.

## Acknowledgements

We would like to thank all members of the Berkeley Par Lab OS group and several people at Lawrence Berkeley National Labs for their valuable contributions to the ideas in this paper.

## References

- [1] T. E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1991.
- [2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [3] K. Asanovic, D. A. Patterson, Z. Tan, A. Waterman, R. Avizienis, and Y. Lee. Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Proceedings of the 4th Workshop on Architectural Research Prototyping (WARP)*, 2009.
- [4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.
- [5] P. Barham et al. Xen and the art of virtualization. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 2003.
- [6] A. Baumann, P. Barhamy, P.-E. Dagandz, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhanian. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.
- [7] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 158–170, New York, NY, USA, 1994. ACM.
- [8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.
- [9] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.
- [10] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*, 2008.
- [11] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *in Proceedings of the Fourteenth ACM symposium on Operating Systems Principles*, pages 189–202, 1993.
- [12] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1995.
- [13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 1995.
- [14] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
- [15] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to os interference

- using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.
- [16] A. Gupta and D. Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Trans. Netw.*, 3(5):501–508, 1995.
- [17] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. Technical report, Stanford, Stanford, CA, USA, 1991.
- [18] Intel. Intel threading building blocks, 2010. <http://www.threadingbuildingblocks.org>.
- [19] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 10, Washington, DC, USA, 2003. IEEE Computer Society.
- [20] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.
- [21] C. Lameter. Extreme high performance computing or why microkernels suck. In *In Proceedings of the Ottawa Linux Symposium*, June 2007.
- [22] R. Liu, K. Klues, S. Bird, S. Hofmeyr†, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client os. In *HotPar '09: Proc. 1st Workshop on Hot Topics in Parallelism*, March 2009.
- [23] W. L. Lynch, B. K. Bray, and M. J. Flynn. The effect of page allocation on caches. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 222–225, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [24] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel scalability of video decoders. *J. Signal Process. Syst.*, 57(2):173–194, 2009.
- [25] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1994.
- [26] A. K. Mok, X. A. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 75, Washington, DC, USA, 2001. IEEE Computer Society.
- [27] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6–16, May-June 2008.
- [28] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.
- [29] H. Pan, B. Hindman, and K. Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proc. of HotPar*, 2009.
- [30] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.
- [31] J. H. Saltzer. Traffic control in a multiplexed computer system. Technical report, MIT MAC-TR-30, July 1966. p. 29, <http://www.bitsavers.org/pdf/mit/lcs/tr/MIT-LCS-TR-030.pdf>.
- [32] Z. Tan, A. Waterman, Y. Lee, R. Avizienis, H. Cook, K. Asanovic, and D. Patterson. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *DAC '10: Proceedings of the 47th Annual Design Automation Conference*, 2010.
- [33] V3VEE. Palacios: An OS Independent Embeddable VMM. <http://www.v3vee.org/palacios/>.
- [34] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 181–192, New York, NY, USA, 1998. ACM.

- [35] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *SOSP '03*, 2003.
- [36] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schausser. Active messages: a mechanism for integrated communication and. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1992.
- [37] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Proceedings of the Linux Clusters: The HPC Revolution conference*, Champaign - Urbana, IL, June 2001.