# Thread Migration in a Replicated-kernel OS

David Katz
Johns Hopkins University Applied Physics Laboratory
david.katz@jhuapl.edu

Antonio Barbalace, Saif Ansary,
Akshay Ravichandran and Binoy Ravindran
ECE Department, Virginia Tech
{antoniob, bmsaif86, akshay87, binoy}@vt.edu

***Abstract*—Chip manufacturers continue to increase the number of cores per chip while balancing requirements for low power consumption. This drives a need for simpler cores and hardware caches. Because of these trends, the scalability of existing shared memory system software is in question. Traditional operating systems (OS) for multiprocessors are based on shared memory communication between cores and are symmetric (SMP). Contention in SMP OSes over shared data structures is increasingly significant in newer generations of many-core processors. We propose the use of the replicated-kernel OS design to improve scalability over the traditional SMP OS. Our replicated-kernel design is an extension of the concept of the multikernel. While a multikernel appears to application software as a distributed network of cooperating microkernels, we provide the appearance of a monolithic, single-system image, task-based OS in which application software is unaware of the distributed nature of the underlying OS. In this paper we tackle the problem of thread migration between kernels in a replicated-kernel OS. We focus on distributed thread group creation, context migration, and address space consistency for threads that execute on different kernels, but belong to the same distributed thread group. This concept is embodied in our prototype OS, called Popcorn Linux, which runs on multicore x86 machines and presents a Linux-like interface to application software that is indistinguishable from the SMP Linux interface. By doing this, we are able to leverage the wealth of existing Linux software for use on our platform while demonstrating the characteristics of the underlying replicated-kernel OS. We show that a replicated-kernel OS scales as well as a multikernel OS by removing the contention on shared data structures. Popcorn, Barrelfish, and SMP Linux are compared on selected benchmarks. Popcorn is shown to be competitive to SMP Linux, and up to 40% faster.**

***Keywords*—replicated-kernel OS; Linux; thread migration;**

## I. Introduction

As chip manufacturers continue to grow the number of cores per chip to fulfill the demand in computational capacity that the market has come to expect, researchers are questioning the scalability of traditional SMP operating systems [1], [2], [3]. These scalability issues are caused, mostly, by cache coherence protocols. Coherency is not easy to scale in high core-count processors [4], so researchers are investigating new solutions [5]. This scalability problem is exacerbated by the fact that we are close to hitting the power wall [6]. Per-chip power dissipation is now limiting the complexity of cores and the on-chip interconnect that can be deployed. This limitation necessitates the use of simpler cores and low-power intra-chip interconnects.

The computing landscape is in a transition phase. Silicon photonics is a promising solution for most of these problems [5] but is not yet viable, whereas cache coherent many-core processors with hundreds of cores are already commodity hardware, e.g., Tilera TileGx [7], Intel Xeon Phi [8]. Results on such processors show that cache coherency can be made to scale – but not as well as on low core-count multicore processors (see Figure 1).

The question of which operating system design scales better on emerging high core-count hardware has many answers. Despite the undisputed success of the monolithic task-based SMP operating system design, which is utilized by most traditional OSes (such as Linux, UNIX, Solaris, Windows, etc.), researchers argue that new operating system models that deviate from that design are needed [9]. Examples of these new designs include Barrelfish [2], FOS [3], Corey [1] and K42 [10]. Corey and K42 question the internal organization of traditional OSes for SMP, proposing a from-scratch implementation. Some of the ideas that they propose, like per-core data structures, have been successfully implemented in traditional OSes. More recently, Linux, as a representative of the SMP OS class, has been demonstrated to scale on multicores to a bounded number of CPUs through the use of expedients like RCU, Sloppy Counters [11], scalable locks [12], and BonsaiVM [13]. These efforts have provided hope that the use of the traditional SMP OS paradigm, to which programmers are accustomed, does not have to be disrupted in favor of a vastly different system.

Scalability problems are also evident at low core counts. Figure 2 reports how the Linux memory subsystem fails to scale when running a real-world benchmark, NPB IS [14], because of contention on shared data structures. Much of the work that addresses the problem of making Linux scale better on shared memory platforms focuses on how to reduce contention on shared data structures. However, other problems also affect OS scalability such as TLB shootdown [3], [1] and non uniform memory access.

Unlike point-wise research on SMP OS scalability, previous research on multikernel OSes shows that most of the scalability problems can be addressed by alterations to the systems model itself. In this paper we propose extending the multikernel OS principles from a multiserver-microkernel OS model to a monolithic task-based OS model. We augment the multiserver-multikernel OS model with distributed OS concepts and introduce the replicated-kernel OS design. Our replicated-kernel OS design presents the exact same programming interface to the user as a traditional SMP OS. In our replicated-kernel Linux realization, common Linux applications can be made to run without any modifications. Within the kernel-layer, however, operating system functionality is distributed across kernel instances transparently to user-space applications. Our replicated-kernel OS is a multikernel OS implemented with traditional SMP OSes as the kernel building block. Each kernel instance is a self-standing SMP Linux OS

running on a subset of the available cores in a many-core processor, is given exclusive access to a region of memory, and interacts with other kernel instances cooperatively. Through cooperation, the kernel instances together form a single consistent operating environment in which user-space applications run. Leveraging message passing, many distributed OS features such as a distributed namespace [15] and distributed virtual shared memory [16] are implemented.

In this paper we present the replicated-kernel OS model and tackle the problem of thread migration between kernels in a replicated-kernel OS. Thread migration enables mulithreaded applications to run their threads in parallel and on different kernels in a replicated-kernel OS. We focus on distributed thread group creation, context migration, address space consistency and synchronization for threads that execute on different kernels but belong to the same distributed thread group. The problems of thread migration and address space consistency, in the context of a distributed OS, have never been considered together before [17], [16]. Moreover, these problems were not considered by the multikernel OS research providing remote thread creation as an alternative to thread migration [2], or by research considering client-server communication as the ultimate solution [3]. With this paper we answer the question of whether it is possible to extend the multikernel model to monolithic OSes and if such an extension scales as well as multikernels and SMP OSes on multicore processors. Results show that it is possible, but scalability is application dependent. A comparison of Popcorn, Barrelfish and SMP Linux on selected benchmarks demonstrates that Popcorn is competitive with Linux, and up to 40% faster. Popcorn is additionally faster than Barrelfish for two out of three cases, providing speedups of up to 17x.

We have realized a prototype of this design, called Popcorn Linux, which runs on SMP multicore x86 machines. We believe that because this model has been successfully applied to Linux, it can also be applied to other OSes. Popcorn is the first in this category that uses an SMP OS as the base for a multikernel, marrying concepts from distributed OS literature to feature-rich, well supported and ubiquitous SMP OSes. By implementing inter-kernel thread migration in Popcorn Linux, we demonstrate that thread migration can be achieved seamlessly in a non-SMP OS while continuing to provide the benefits of the idioms user-space programmers have come to enjoy. Popcorn Linux is the result of a significant re-engineering effort and is publicly available at http://www.popcornlinux.org.

In Section II we present related works. In Section III we introduce some basic OS definitions and summarize Linux's task and memory management subsystems. In Section IV we describe the replicated-kernel OS model and we present an overview of Popcorn Linux. Section V presents the thread migration, the address space consistency, and the user-space synchronization mechanisms. Section VI and Section VII presents the experimental setup and the results of the experiments and related discussion. We conclude in Section VIII.

## II. Related Work

OS Clustering [18] is the most similar work to Popcorn Linux. It strives to investigate a "middle ground" between the multikernel OS design and a scalable monolithic OS for emerging high core-count multiprocessors. Like Popcorn, OS Clustering is implemented within the Linux environment, and

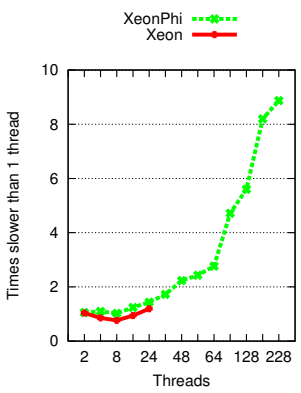



Fig. 1. Overhead to access a shared data structure with increasing number of threads using shared memory on different processors. Run on Intel Xeon and Intel Xeon Phi.

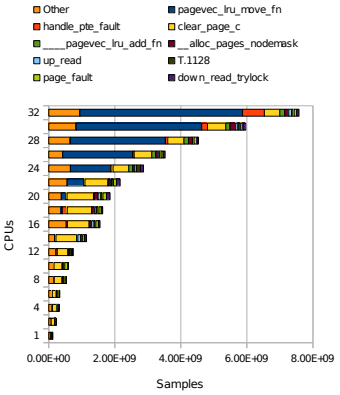



Fig. 2. SMP Linux kernel overhead (NPB IS OpenMP) traced with the Linux perf utility on the main thread. Run on AMD Opteron machine presented in Section VI.

its goal is to provide the familiar POSIX programming interface and shared-memory programming model. However, OS Clustering makes use of a virtual machine monitor (VMM) to make multiple kernels coexist on the same multicore hardware, and coordinate system services among different kernels, e.g., address space consistency. In Popcorn there is no VMM. Following the multikernel OS design we did not add any piece of software between the hardware and the OS. The risk with a VMM is that it relies on shared memory variables for inter-core communications, re-inserting the contention on shared data structures at another software level. Before OS Clustering, Hive [19] implemented the idea of partitioning cores of a multiprocessor to different OS kernels. This work was based on IRIX UNIX, and was implemented on research hardware. Each group of cores that loads a single kernel is referred to as a Cell. Kernels strictly communicate with remote procedure calls (RPC)s. User-space applications are able to spawn on varying kernels and exploit hardware shared memory. Popcorn, as well as Hive, exploits a UNIX-like operating system as the base for a replicated-kernel OS. However, Popcorn targets commodity hardware.

Barrelfish [2] introduced the multikernel OS design based on the microkernel OS model. In this design each multiprocessor core executes a microkernel and its application stack. Microkernels communicate by message passing, rather than exploiting the abundant shared memory available on commodity multiprocessors. This is called a shared-nothing approach between kernels. The multikernel OS re-applies the microkernel principle of splitting the kernel, but splits the microkernel objects between cores. Like Barrelfish, Popcorn runs multiple instances of the same kernel on a multicore. However Popcorn is different in that it is based on a monolithic kernel. Instances communicate strictly via message passing. FOS [3] is another multikernel OS specifically designed to run on thousands of cores. It factors operating system functionality into a number of services and focuses on their spatial subdivision within a thousand-core processor. It applies distributed system principles to the multikernel, multiserver-microkernel OS design. Kernel services and applications run on different cores. Kernel services are replicated throughout the processor on different cores to allow application processing

cores to make use of services provided by the "closest" OS service cores. Routing to spatially local service cores is central to FOS. Task-to-task and kernel-to-kernel communication is entirely based upon messages. In Popcorn, kernel-to-kernel communication is also message based, but task-to-task is not implemented since Popcorn facilitates memory sharing for user-space components. Kernel services in Popcorn normally run on the same core as the calling task.

Plan 9 [15] is a distributed OS that introduces a single execution environment across a network of computers. Sprite [20], MOSIX [21] and more recently Kerrighed [17], enable a single execution environment over a network of computers with process migration. All of them are UNIX-like solutions – there is a MOSIX port for Linux and Kerrighed is based on Linux. All of these address a network of computers and exclusively consider process migration. Popcorn implements a single execution environment among different kernels on a single multiprocessor machine allowing for process and thread migration.

## III. Background

Linux[1] is a UNIX-like operating system that evolved from an OS for single processor systems to an OS for multiprocessors. Because of its open-source nature, and the world-wide popularity it has gained, it has been the target of many works that have addressed scalability on multiprocessors [11], [12], [13]. Linux is a multitasking, monolithic OS based on shared memory. Both kernel space and user space tasks communicate through shared memory, and Linux extensively exploits hardware-provided cache-coherent shared memory.

In this work we refer to tasks in the Linux terminology. A task is a kernel scheduler entity that is described by a `struct task_struct` data structure. A task can be a user-space thread or a kernel-space thread, *kthread*. A process in Linux terminology is a group of threads, called a *thread group*. In a Linux thread group, tasks share the same address space: they share memory, and therefore the same address space descriptor, `struct mm_struct`. All kernel threads share the address space descriptor of the kernel, `init_mm`. A thread group's tasks do not only share the same address space but also file descriptors, user privileges, namespaces, signals, locking constructs, etc.

Linux is a task-based (or process model [22]) monolithic OS. Task identity and attributes (such as its process identifier, scheduling priority, address space descriptor, etc.) are maintained when its execution switches between user-space and kernel-space. However, there is a separate task execution context, i.e., CPU registers and stack, per privilege level (user and kernel). The kernel-level task stack is maintained for the lifetime of a syscall, while the user level stack is maintained for the lifetime of the task. When the task issues a synchronous call to the kernel, a kernel execution context is created and its user context is saved in its `struct task_struct`. The user execution context is restored when execution returns to user-space.

In Linux, tasks are able to migrate between CPUs. This helps to maximize hardware utilization. Migration is eased by shared memory. A single address space descriptor is pointed to by all cores on which the application is running. Cores access this data structure, synchronized by a lock.

[1]All references to Linux in this paper pertain to Linux kernel 3.2.14

### A. Linux Task Management

Task creation and deletion are managed by the OS, and the OS scheduler decides when each task is run. The schedulability of a task depends on the state in which it resides. The task state can be either running (eligible for execution), interruptible (not eligible but can be awoken by a signal), uninterruptible (not eligible and cannot be awoken by signals), stopped or traced (it is being debugged).

All tasks in Linux belong to a hierarchy. All tasks are children of some other task except two. The *init* task is the parent of all user-space tasks. The *kthreadd* task is the parent of all kernel-space tasks. A new task in Linux is created by forking from the parent task. A fork operation consists of creating a duplicate of the calling task. Once the new task is created, it can execute independently of the parent task or in the same thread group, depending on the options used when calling fork. A user-space task forks by calling the *fork* syscall. Alternatively, a kernel-mode task calls the `kernel_thread` function to create a new kernel task. Regardless of user-space or kernel-space invocation, task creation is handled through `do_fork`, which implements the fork operation. When the caller is a user-space task, the child process begins execution at the point after the fork syscall was invoked. When the caller is a kernel-space task, the child begins execution from a function that is passed to `kernel_thread`. The `do_fork` function clones all of the parent task's data structures for use in the child's newly created `task_struct` object. The items that are optionally shared with the parent task include locks held, file descriptors, signal handlers, its memory map, the namespaces to which it belongs, its IO, etc. Reference counting is used to account for these shared objects.

A task exits when it receives a signal to exit, experiences an error such as a segmentation fault, or when the task invokes the exit system call. When one of these events occurs, the kernel responds by invoking the `do_exit` function from the task's context. This function arranges for the removal of kernel objects associated with the exiting task. Reference counters associated with the tasks shared objects are decremented and those objects are destroyed if the count reaches zero.

### B. Linux Futex

User-space synchronization mechanisms that exclusively use shared memory by spinlocking, are not efficient. This is because a spinlock starves other tasks of valuable processing time. Therefore the Linux community introduced the futex, a fast user-kernel space mutex [23]. Futexes perform *wait* and *wake* operations involving the OS in the slow path only. The OS can then schedule other threads during the wait. When the thread performs a *wait* operation, Linux atomically enqueues it in a per futex queue, which contains all of the waiters for that futex. When the current futex owner releases it, the next thread in the queue takes ownership and resumes execution.

### C. Linux Memory Management

We refer to an address space as a memory map combined with that memory's contents. Linux is a virtual memory OS. Hardware paging is exploited to allow any process to see the entire address range (with the exception that user-space cannot access the kernel-space range). The hardware MMU translation mechanism issues a page fault when a virtual address is accessed that does not have a physical address associated with it. This triggers Linux's page fault handler,

do_page_fault. The virtual-to-physical address translation is saved in hierarchical page tables. The processor can point to one table at the time, which Linux saves in the pgd field of the currently active mm_struct. When a task switch occurs on a CPU, the MMU is updated to point to the page table associated with the new task, changing the active virtual address space. Each page table entry (PTE), which indexes a single virtual page, contains information about how that memory can be used. This is how caching behavior and access permissions are specified for a given virtual page.

To efficiently exploit memory resources, Linux implements on-demand paging. When an application maps a new region of memory into its address space, the physical pages backing the new virtual memory area are not allocated immediately. Rather, the kernel creates a record of the fact that it owes that process memory in a data structure called struct vm_area_struct (VMA) and enqueues it in the process's VMAs list. When the application first accesses that memory it causes a page fault that triggers the kernel to select a physical page for the faulting virtual page, and update the page tables accordingly. To select the physical page, the kernel consults the list of VMAs for that task. A VMA can span multiple pages and can either refer to bare memory (anonymous-mapping), or can be file backed. During the virtual to physical mapping of a file-backed page, the content is fetched from the file and written to the selected page in memory. In order to further optimize the usage of the memory, Linux also implements copy on write (COW). When a process is forked, writable pages in the parents address space are shared with the child, but are write protected. When the child attempts to write to such a page, a page fault occurs due to the write protection, and that page is copied to a newly selected page. The virtual to physical mapping is also updated such that the child's page now refers to the new page, the page is marked writable, and the child can now write to that page. COW pages and zero pages (malloc) are called special pages.

In Linux, a process's memory layout can be manipulated through the use of various APIs, including mmap, munmap, mprotect, and mremap. All of these modify shared data structures (e.g., struct mm_struct and struct vm_area_struct). mmap, munmap and remap allow for adding, removing, modifying an area of memory in the task's address space, respectively. mprotect allows for changing a memory area's protection flags.

## IV. Replicated-kernel OS

The replicated-kernel OS design merges ideas from both multikernel OSes and distributed OSes. The replicated-kernel OS strives to provide a single system image, typical of SMP and distributed OS, on top of an operating system made up of multiple kernels. Figure 3 shows this design. On multiprocessor hardware, kernels run on a single CPU or on a group of CPUs, and each kernel instance is given a partition of the hardware resources (e.g., RAM and cores). Kernel instances do not share memory (they must stay within their memory partition), but communicate exclusively and explicitly by messages, as in a multikernel OS. This messaging is used to stitch together a single-system image to host user applications. The fact that applications run on a replicated-kernel layer is transparent to the application, as work is done at the kernel layer to emulate an interface that matches the traditional Linux environment. Kernels constituting a replicated-kernel OS are not exact

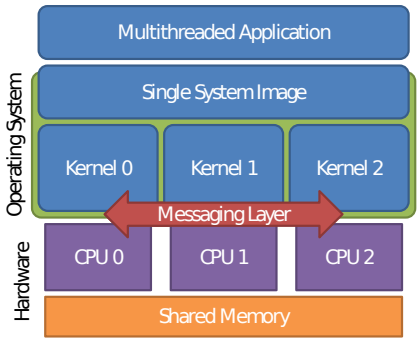



Fig. 3. Replicated-kernel OS design. Hardware resources are partitioned among kernels. Multithreaded applications transparently run among different kernels, and therefore different hardware partitions.

replicas of one another. Rather, the kernel state is only partially replicated: each kernel manages the resources that are allocated to it. Total replication is unnecessary, and in fact undesirable from a performance perspective. The overhead of replicating data structures is not negligible and the replicated-kernel OS must replicate only what is absolutely necessary while also maintaining logical consistency across kernel instances. One example of this concept is the page-tables associated with a process with threads executing on disparate kernel instances. The mechanism required to maintain exact and complete replicas of a process's page table would be prohibitively high overhead, without providing any benefit (since not all threads need the entire address space). In most cases, it is sufficient to implement a partial replication. In the future we envision this model being applied not only to symmetric multiprocessor platforms but also to emerging heterogeneous platforms, where cache coherency among processors is an option.

### A. Popcorn Linux

Popcorn Linux implements the replicated-kernel OS design [24]. The replicated-kernel OS mechanisms deployed in Popcorn are completely transparent to user applications. They give an application the illusion of running in an SMP Linux environment.

*1) Boot and Resource Partitioning:* Popcorn Linux boots one Linux kernel per core or per group of cores on multicore Intel x86 machines. Popcorn required a non trivial re-engineering of the Linux kernel and allows for multiple Linux kernel instances to share hardware resources of a single machine without virtualization. Kernels are peers. Each kernel instance has full control of a set of CPUs, a part of physical memory, and a subset of the available peripherals. Kernels have a unique identifier, and boot sequentially. When a kernel joins the replicated-kernel it broadcasts information describing its resources to all other kernels. This allows those resources to be shown in /dev and /proc on all kernels, if the resources are devices, CPU or memory. This functionality is part of the single system image. The kernel identifier that is chosen for a kernel instance is the smallest logical ID of the CPUs assigned to that kernel.

*2) Messaging:* Linux is not a message-passing OS, though task-to-task messaging is implicitly possible using shared memory (POSIX message queues). However, there is no concept of message-passing in kernel-space. Because a replicated-kernel OS relies on message passing as its fundamental coor-

dination building block, it was necessary to extend Linux with a messaging capability. Popcorn introduced an inter-kernel and intra-kernel task-to-task kernel-space messaging layer.

Messages in Popcorn Linux are comprised of structured data that is recovered intact by the recipient. Message passing is implemented on top of shared memory, and is one of the only places Popcorn must break the no-sharing invariant. Notification of message availability is accomplished through the use of interprocessor interrupts (IPI). As in [2], we combine IPI with polling to reduce IPI overhead.

To solve the problem of task addressing, each kernel instance is given a non-overlapping range of TIDs to assign to tasks upon creation. When a task migrates to a different kernel instance, its TID migrates with it, and the task's home kernel instance is notified of the migration. To find the current location of a task for message addressing, the target task's TID is first consulted to determine that task's home kernel instance. The home kernel instance is then asked for the task's current kernel of execution.

*3) Single-System Image:* On top of kernel space lives a single operating environment for user-space applications. All of the separate kernel instances work together to present an interface that behaves exactly like Linux. Applications are unable to tell the difference between an SMP Linux and Popcorn Linux system. Because of this, Popcorn is able to leverage the extensive available library of robust and mature Linux applications. The single-system image allows for not only static resource enumeration (e.g., `/proc/cpuinfo` enumeration of all the CPUs in the system) but also for dynamic usage of resources (e.g., enumeration of the current running tasks in the system). All kernels are aware of all tasks that exist in the system and what they are doing. This allows any kernel to send signals to any task on any other kernel. Inter-process communication is possible between processes and threads that exist on different kernels as well. Popcorn facilitates the use of shared memory, semaphores, message queues, mutexes and futexes across kernel-instance boundaries. Popcorn Linux works to support those constructs regardless of where they are originally instantiated or used. In Popcorn each kernel owns a partition of the available machine's resources. In a multicore machine, resource partitioning can be changed at runtime. This allows for flexibility in handling resource-specific operations. For example, an application that needs to access the CDROM can either be migrated to the kernel that owns that device or it can request that kernel control of that resource be transferred to its current kernel. Unlike in a distributed setup, the device driver can be migrated. Popcorn also allows proxying of device access (e.g., APIC device [24]).

Popcorn does not currently implement a distributed scheduler. Each kernel schedules its own tasks independently.

## V. Thread Migration

A thread migration involves the coordination of two kernel instances to transfer a thread and its state from one instance to the other. The thread that is migrated ceases to execute on the originating kernel instance, and resumes execution on the receiving one. Thread migration targets user-space tasks exclusively, i.e., kernel threads do not migrate. Once migration occurs, work must be done to support its continued execution on the new kernel instance. This work includes the migration of resources that the thread has already allocated. These resource migrations can occur either at the time of the initial migration (e.g., register values) or on-demand (e.g., page table entries). Additionally, replicated data structures must be maintained as they evolve, because many of these resources (e.g., page tables) are shared between multiple threads through replication and available simultaneously on different kernel instances for use. A number of distributed algorithms, some custom and some well established, were employed to replicate and maintain consistent shared data structures across kernel instance boundaries. Some of those mechanisms will be discussed below, including address space migration and futexes. Initial thread migration and thread life-cycle maintenance will also be described.

We propose a thread migration mechanism in which a thread's resources are migrated on demand. However, a bulk migration implementation has also been developed for completeness, and can be enabled at compile-time.

### A. Migration Mechanism

The thread migration mechanism implemented in Popcorn Linux supports explicit application-controlled migration only. The application invokes the POSIX `sched_setaffinity` syscall to request a migration. When a thread migrates away from a kernel instance, the data structures that represent it on the originating kernel instance are not destroyed. Rather, the thread on that kernel instance remains dormant and is considered a *shadow task*. Shadow tasks serve a number of purposes. They are custodians of the resources allocated by the thread (such as memory) while it executed on that kernel. Those resources can be migrated or otherwise used at some later point. The presence of shadow tasks associated with their resources keeps reference counting resource deallocation from occurring in many cases. Another use of a shadow task is to remove the necessity to recreate the thread's data structure in the event that the thread migrates back to that kernel instance. When that occurs, the thread can simply re-claim its shadow task, instantly gaining access to all of the resources that the shadow task was maintaining on its behalf.

*1) Distributed Thread Group:* When the first thread of a process leaves a kernel, the thread group is mutated into a distributed thread group (dtgroup), and a distributed thread group identifier is added in the `struct task_struct`. This distributed thread group identifier is used to track which threads are members of a given distributed thread group. All thread migrations are initiated by the originating kernel instance. A subset of the thread state data is communicated to the receiving kernel instance, which includes user-space memory layout, execution context (i.e., user-space CPU register values), task identifiers, user credentials, scheduling and priority information, and the distributed thread group identifier. The user-space memory layout information that is transferred includes the address ranges of the stack, heap, environment variables, program arguments, and data segments. When a thread is migrated to a kernel on which it has never executed before, the receiving kernel does not have an existing shadow task to host the received thread. In that case, a new task is created to host the migrated thread's state. In order to create the new task, a new kernel thread is first created. That new kernel thread is then evolved into a user-space thread, and the thread state that was transferred is installed in it. If the receiving kernel instance hosts other threads in the migrating thread's distributed thread group, the newly migrated thread must be placed in the same local thread group as those other threads.

Because threads that belong to the same thread group share many OS resources, including their signal handlers, memory map, and file descriptors, the newly migrated thread is set up to share them. If there are no existing thread group members on the receiving kernel instance but previously there were, a dangling memory mapping `struct mm_struct` is still present on that kernel and will be installed in the thread. This is discussed in more detail later. When a thread is migrated back to a kernel instance on which it has previously executed, the hosting thread already exists in the form of a shadow task. That shadow task is also already part of the correct thread group, and shares signals, signal handlers, and open file descriptors with the rest of its thread group. These objects do not need to be altered in this case but the thread's current state information, which was sent from the originating kernel instance, must be installed into the shadow task.

*2) Exit:* A thread can migrate any number of times, and can therefore have a shadow task on any kernel instance of the replicated-kernel OS in addition to the kernel on which is currently running. When that thread exits, every shadow tasks must also exit. Popcorn notifies all kernels when a thread of a distributed thread group has exited. When shadow tasks receive an exit message, they execute `do_exit`.

When threads of a distributed thread group exit, they do not always free their resources, because it is possible that these resources will be needed again when another thread migrates to that kernel instance. Resources that are reference counted have their reference counts artificially increased by one, and maintained in a list for safe keeping. These resources must be deallocated when the thread group exits. To accomplish this, when a thread exits, it checks to see if it was the last thread (including threads executing on remote kernel instances) to exit within the distributed thread group. If the exiting thread is the last, it sends a message to all remote kernel instances indicating that the thread group is closing. At that point, all kernel instances can free all resources for that distributed thread group. Another reason for maintaining resources while no threads are executing is that some resources, such as the thread group's memory map (held in a `struct mm_struct` data structure), are partially replicated. It is therefore unknown if a given resource is the only one with some critical information, such as mappings in the case of a memory map. Deallocating that data would risk removing the only copy of that information, and removes the possibility that it may be resolved at some point in the future should a remote thread group member need it. It is therefore necessary to hold all resources associated with a distributed thread group until all threads in that thread group exit. It is important to note that resources being reserved still undergo consistency maintenance to ensure that those resources are up to date with respect to their replicated counterparts on remote kernel instances.

### *B. Address Space Migration Mechanism*

A process' address space is contained in a series of sparse data structures. Because kernel instances do not share kernel memory, those data structures are replicated per-kernel. Popcorn implements partial replication, through a protocol designed to maintain consistency between replicas (there is previous work, such as [1], in this space). The address space of a process can grow, shrink and be modified based on the memory requirements of the process. As this address space change happens, Popcorn must take action to ensure that local changes are reflected globally to keep the replicated address space consistent across kernel instances. This does not imply that it is necessary that all replicated memory maps associated with the same distributed thread group on all involved kernel instances be exactly the same, because not all thread group members need the entire address space at all times. Rather each kernel instance needs to provide enough of the address space to the threads that it is hosting to support their execution, while also ensuring that invalid memory mappings are never available. Additionally, it is necessary that analogous mappings on different kernel instances match exactly. Two mappings are consistent with one another if and only if the following attributes match: virtual page, physical page, protection status, backing source (anonymous or file), and special page status (COW, zero, normal).

*1) On-Demand Migration:* With on-demand address-space migration, an empty memory map is created for a thread when it migrates to a kernel which none of its distributed thread group members has visited before. Once the thread executes it begins to cause page faults because the memory map was not populated. Popcorn alters Linux's memory handling subsystem to catch page faults and adds a step at the beginning of the fault handler to attempt to retrieve mappings from remote distributed thread group members. This ensures that mappings are migrated only as they are needed by an executing thread.

When a page fault occurs, a query is sent to all kernel instances specifying the distributed thread group identity, and the faulting virtual address (*dtgroup_query_mm*). The remote kernel instances that receive this query search for any threads that they may be currently hosting that are members of the faulting thread's distributed thread group. If one is found, its `struct mm_struct` is used to resolve the mapping. If for any reason after searching existing threads and saved memory maps, none is found, a response is sent that indicates that no mapping was found on that kernel instance. Otherwise, all information about the found mapping is sent as a response.

Special treatment is applied in cases where the resolved mapping is COW. When a mapping request results in a mapping that is a COW page, the responding kernel instance first breaks the COW by allocating a new page and copying the original content, then retries the mapping search before responding to the requesting kernel instance. This is necessary in order to ensure address space consistency. If the COW mapping was instead retrieved without first breaking the COW page, the newly installed mapping on the requesting kernel instance would also be mapped COW. If code on both kernel instances then breaks the COW, two different physical addresses would be assigned to the same virtual address, which breaks consistency. It was decided to pay the performance penalty associated with breaking the COW prior to migration to avoid this situation.

Because the address space is partially replicated, responses gathered from kernels may have different content. Kernel instances can either indicate that no mapping exists; a valid `struct vm_area_struct` exists, but no physical page has been allocated; or a complete and specific virtual-to-physical page mapping exists. If a message is received with a complete virtual-to-physical page mapping, it is given precedence. If no such message is received, messages indicating that the virtual memory area is valid are given precedence. If no such messages arrive, then it must be the case that no remote

```
VAR: vaddr (in), taskid (in), VMA (out), PTE (out)
groupid ← thread_group(taskid)
dtgroup_lock((groupid, vaddr))
RESP ← dtgroup_query_mm((groupid, vaddr))
(vma, pte) ← find_map(RESP)
if vma ≠ null then
   if pte ≠ null then
      VMA ← vma; PTE ← pte
      dtgroup_unlock((groupid, vaddr)); exit
   else
      VMA ← vma
   end if
end if
RET ← call(SMP_Linux_faultHandler)
dtgroup_unlock((groupid, vaddr))
if RET is error then
   call(SMP_Linux_segFaultHandler)
end if
```

Fig. 4. Popcorn Linux's page fault handler algorithm. The page fault handler queries the distributed thread group. To protect against multiple distributed page faults on the same address range a distributed lock is taken.

mapping exists for the faulting address.

If no remote mapping is found, the SMP Linux fault handling mechanism is invoked. If instead the `struct vm_area_struct` exists, but no physical page has been allocated to it, then an identical `struct vm_area_struct` is installed in the faulting thread's memory map, and the SMP Linux fault handler is invoked. In that case, the SMP Linux fault handler will map physical memory to the faulting address. If instead, both a `struct vm_area_struct` and a physical page mapping are reported to exist, an identical mapping is installed in the faulting thread's memory map. Unlike the other cases, both a `struct vm_area_struct` and `struct pte` are now assured to exist, so the SMP Linux fault mechanism does not need to be invoked. If neither a local mapping nor a remote mapping is found, the Linux fault handler will ensure that a segmentation fault is reported.

*2) Concurrent Mappings:* On-demand address space migration creates a number of concurrency challenges due to mapping retrieval interleavings between kernel instances. Suppose two threads in the same distributed thread group running in two kernel instances fault on the same virtual address at the same time and no kernel has any mapping for that virtual address. In that case, neither of those kernel instances will receive any responses that include a mapping. Each will then default to the normal Linux fault-handling mechanism, wherein a new physical page is assigned to the faulting virtual page. But those physical addresses will be different. This results in a distributed address space in which there are two physical pages mapped to the same virtual page, which is an inconsistent state. This example is representative of a class of similar consistency failures that Popcorn addresses. In order to remove this class of failures, Popcorn implements distributed mutual exclusion to allow consistent concurrent mapping operations of a page or group of pages (the same communication cost applies in both cases).

We implement a variation of Lamport's distributed mutual exclusion algorithm in which a Lamport queue is lazily-created for every distributed thread group/faulting virtual page pair (*dtgroup_lock*, *dtgroup_unlock*) in Figure 4). This ensures that Popcorn can concurrently resolve mappings for threads in different thread groups, or threads that are in the same distributed thread group but are faulting on different virtual pages. In order to minimize the memory requirements for this mechanism, queues are created dynamically as they are needed, and are destroyed when they are no longer needed. The mutual exclusion algorithm requires that all nodes share a common notion of time. A memory counter that is shared between kernel instances was created to serve as a logical time-stamp. A single shared page is used to host a counter which is fetched and incremented each time a time-stamp is required.

*3) Operations on a Group of Pages:* Certain types of address space modifications require a more coarse grain lock than the per-page Lamport lock used for mapping retrieval operations. To address this need, a single lock was created for each thread group per kernel. When a `munmap`, `mremap`, `mprotect`, or `mmap` operation is executed against a region of memory, such a lock is secured. The addition of this lock removes the overhead of a naive implementation which would require creating a Lamport queue for every virtual page in the address space. We call this lock a *heavy* lock. A heavy lock behaves exactly the same as one of the fine-grained per-page locks, with a slight variation. When an entry is added into the heavy queue, the same entry is also added into every existing fine-grained per-page queue, in order of time-stamp. When an entry is removed from the heavy queue, the corresponding heavy entry is removed from every fine-grained per-page queue. Additionally, when a new per-page queue is created, one entry is added into that new queue for every entry in the heavy queue, also ordered by time-stamp. A heavy lock is not acquired until its corresponding entry is at the front of every queue for the process, including the heavy queue and all per-page queues. This is conceptually equivalent to creating a queue for every page in the user address space. This eliminates the overhead that would be necessary to create queues for every virtual page in the address space. This mechanism coexists with and interacts appropriately with the per-page distributed mutual exclusion implementation for mapping retrieval. The exact mechanism implemented for each address space modification event and its implementation is covered in detail in [25].

*4) Prefetching:* Mapping prefetch was implemented to reduce the number of mapping retrieval messages that must be circulated to maintain the address space. The message that carries mapping responses was extended to include a configurable number of prefetch slots. Each slot includes the start and end address of a contiguous region of virtual-to-physical mappings. By describing mappings in this way, large quantities of mappings can be described in a compressed format. Popcorn tries to fit as many mappings into each response as there are slots in the prefetch message. If there are not enough mappings in the `struct vm_area_struct` to fill the slots, the remaining slots are left empty, without additional overhead.

### C. Inter-Thread Synchronization

The *glibc* library relies heavily on futex. Futex was ported to Popcorn to support applications that use *glibc*. We extend futex to Popcorn by adopting a client/server model. The server kernel, which is the kernel that the lock's physical page belongs to, maintains the futex queue. This enforces serialization. All other kernels are clients and must send a message to the server to be enqueued when the application calls *wait*. *wake*

operations are similarly directed to the server kernel. That kernel receives the request and wakes up the first task in the queue, notifying it with a message. Upon receiving this message, the new lock owner resumes execution.

## VI. Experimental Setup

All results were collected on a multiprocessor x86 64bit, 32-core server machine, assembled with two AMD Opteron 6274 CPUs running at 2.2GHz with 128 GB of RAM. We compare SMP Linux 3.2.14, Popcorn Linux, and Barrelfish (Mercurial changeset 2249:4211588bbdff). Popcorn is based on Linux version 3.2.14, adding $31k$ lines of code (LOC) to the kernel and $5k$ LOC of user-space tools used for testing and data gathering. Linux was tested with a number of cores equal to the maximum number of threads specified with `OMP_SCHED_AFFINITY`. Furthermore, the kernel command line options `apic=off noacpi` were added to ease data collection. Popcorn Linux was tested with a one core per kernel configuration for a direct comparison with Barrelfish. In all tests power management was turned off to drive the processors to the maximum performance.

Influenced by Clements *et al.* [26], we focused on compute and memory intensive benchmarks.Compute/memory intensive applications from the SNU NPB [14] (IS, CG, and FT), Parboil [27] (LUD), and Rodinia [28] (BFS) benchmark suites were run. These applications were chosen to exhibit diversity in memory access patterns and thread interaction. The benchmark implementations that were used were originally written for OpenMP. They were run with a stripped down version of the OpenMP library, called *pomp*, which is based on a stripped down version of pthread, called cthread. The *pomp* library is an adaptation of Barrelfish's *bomp*, ensuring a fair comparison between OSes.Where specified, the benchmarks were run with *glibc*/`nptl` as fully fledged libc pthread libraries.

## VII. Results

In this section we show the validity of the chosen approach, the cost of thread migration, and the cost of distributed address space consistency maintenance. All results presented are averages of ten runs.

A microbenchmark that quantifies the overhead of a thread migration was also run. The microbenchmark creates a thread and migrates it to a different kernel and back. In Linux, migrating a thread to a different core is fast, being performed on the order of $9\mu s$. In Popcorn the first migration requires up to $2ms$ but a migration back costs only $120\mu s$ (which is still slower than Linux), though it was found that thread migration cost does not necessarily drive workload overhead.

*a) NPB IS, FT, CG:* We selected NPB's IS, FT, and CG applications to compare Popcorn with SMP Linux and Barrelfish. Barrelfish's software distribution includes the class A version of these benchmarks. The class of the benchmark represents the input data size. Class A is fairly small, and highlights system overheads over computational time.

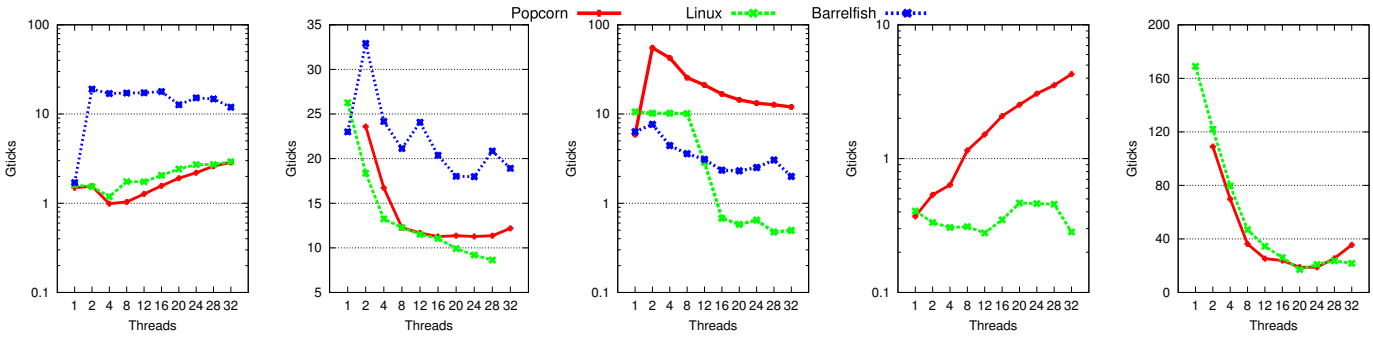
The IS profile is depicted in Figure 5(a). As evident in the figure, the benchmark has scalability issues. Popcorn outperforms competing OSes. Linux is up to $68\%$ slower than Popcorn when using 8 threads, while Barrelfish is up to 17 times slower than Popcorn when using 4 threads. However, beyond 32 cores Linux is slower than Popcorn. This is due to the increasingly significant overhead of Popcorn's messaging layer. A complete analysis of the memory behavior of IS can be found in [25]. Threads in IS have disjoint memory maps. In Linux there is a bottleneck in the memory subsystem characterized by an overhead that is proportional to the core count, which slows the kernel-space execution in IS. The same bottleneck does not exist in Popcorn due to the fact that the replicated-kernel OS capitalizes on disjoint memory accesses in IS, removing contention and allowing for faster execution.

The FT profile is shown in Figure 5(b). In this workload, Popcorn is comparable to Linux. At $8$ cores Popcorn is up to $10\%$ faster than Linux, but Popcorn is up to $30\%$ slower than Linux when using $28$ cores. Barrelfish is up to twice as slow as Popcorn when using a $12$ thread configuration. We traced the FT benchmark in Linux, and discovered that the kernel time is dominated by overhead in the memory subsystem. This time increases with the number of threads. This overhead is again mitigated by Popcorn but the overhead of the distributed protocol and our messaging begins to degrade performance after $16$ cores. Unlike the IS benchmark, the FT benchmark fetches many existing mappings from other kernels, indicating that threads in FT have overlapping memory working sets.

The CG behavior is reported in Figure 5(c). Popcorn significantly under-performs compared to both SMP Linux and Barrelfish. Popcorn is up to $25$ times slower than Linux on $28$ cores, and up to $6$ times slower than Barrelfish on $16$ cores. Barrelfish is the fastest OS up to $8$ cores, in which it is $60\%$ faster than Linux. However, after $12$ cores Linux scales better than any other solution. Tracing reveals that there are no scaling issues in the CG execution on SMP Linux. Time spent in the kernel is constant as the number of threads increases. The memory access pattern in CG is asymmetric, as the main thread shares a piece of memory with every worker thread. In Popcorn, this causes the first kernel to become a performance bottleneck because it is the only one that allocates memory and dispatches mappings to other kernels. Another contributing factor to Popcorn's poor performance is the fact that CG performs five thread migrations for every kernel instance that is used beyond the first kernel instance. The migration cost is significant in this case. This also negatively impacts Barrelfish after $12$ cores.

Why this difference between Popcorn and Barrelfish? According to [2], Barrelfish does not implement on-demand paging. Instead, all memory is allocated at load time. This is a significant difference with Popcorn, and explains Barrelfish's better performance on CG, in which Popcorn is overwhelmed by memory map retrieval traffic. Further analysis shows that Barrelfish was slower than Popcorn in FT and IS due to different services running on each core, for example, for inter-core communication in the gang scheduler.

*b) BFS and LUD:* Rodinia BFS and Parboil LUD do not have Barrelfish ports. Those workloads were run on Popcorn Linux and SMP Linux only. Linux outperforms Popcorn on the BFS benchmark, where Popcorn is up to $15$ times slower than Linux; see Figure 5(d). The BFS benchmark was made to run with the *1M_W* input file. While this input file is the largest shipped with the Rodinia distribution, the benchmark runtime is very short. The performance gains that Popcorn makes over SMP Linux on other workloads are due to the removal of lock contention during accesses to shared data structures. The extremely short duration of the BFS workload negatively impacts Popcorn's performance results, because Popcorn is not given time to compensate for its migration overheads through




(a) NPB IS (b) NPB FT (c) NPB CG (d) Rodinia BFS (e) Parboil LUD

Fig. 5. Execution time of each benchmark on SMP Linux, Barrelfish and Popcorn varying the number of threads. Lower is faster.

reduction of lock contention. This benchmark performs twelve thread migrations for every kernel instance, which contributes to the overhead seen on Popcorn. The analyzed memory behavior of BFS shows a similar access pattern to CG but with a much higher level of locality (see below).

The LUD results are shown in Figure 5(e). Popcorn is up to 27% faster than Linux on 12 cores. As the core count increases, this performance benefit decreases. For the case of 32 cores, Popcorn is slower than Linux by 63%. Our analysis shows that the overhead of the messaging layer is the primary contributor to this slow down. Further analysis of the SMP Linux results reveals that kernel overhead increases with the number of threads. This increase is not solely related to the memory subsystem. The same overhead is not found in Popcorn.

### A. Prefetch

The effect of mapping prefetch is shown in Figures 6(a), 6(b), 6(c), 6(d), 6(e) for IS, FT, CG, BFS, and LUD, respectively. Prefetch exploits data locality to reduce the number of messages in the system. Benchmarks with a locality access pattern similar to FT, BFS, and LUD greatly benefit from prefetch. BFS execution time improves over the one depicted in 5(d) up to 39% at high core-counts. The benefit is more significant when using 4 prefetch slots than 8. This is due to processing time spent acquiring mappings, installing mappings on the receiving kernel, and more time spent transporting the mapping contents between kernel instances. In FT, increasing the number of prefetch slots improves the execution time up to 13% at high core-counts. Prefetching is not always beneficial. For example in IS, the execution time is negatively impacted consistently at any core count by the prefetching mechanism by as much as 5%.

### B. Tailored Consistency

An application specific address space consistency protocol was implemented for the NPB benchmarks. Figure 7 shows the profiles. Knowledge gained through profiling was applied to implement the protocol, reducing the number of messages exchanged. Results show that the new protocol, called *Popcorn Lazy*, improves performance when compared to Linux up to 54% at 18 cores, and over Popcorn implementation up to 42% at 28 cores. This version is always faster than Popcorn, for all NPB benchmarks.

### C. Futex

As discussed above, *glibc* makes extensive use of futex. However, our replicated-kernel OS implementation of futex,

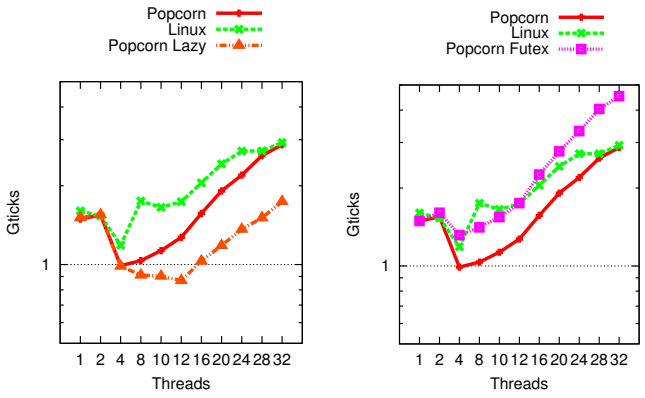



Fig. 7. Comparison of SMP Linux, Popcorn, and Popcorn with Tailored Consistency protocol on NPB IS.

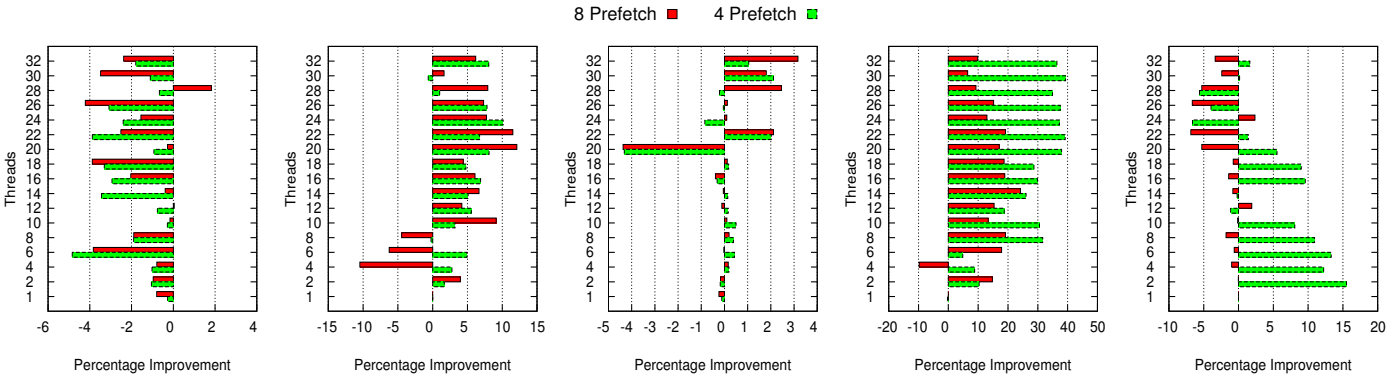



Fig. 8. Comparison of SMP Linux, Popcorn, and Popcorn with futex on NPB IS.

in Popcorn, does not provide better results than spinlocking (*cthread*). This is shown in Figure 8. The Figure shows the NPB IS experiment compiled with *glibc*/`nptl`, i.e., the POSIX threads library. Popcorn with futex is up to 57% slower than Popcorn without futex support, and up to 54% slower then SMP Linux at 32 cores. However, until 12 threads, Popcorn with futex is faster than SMP Linux (up to 20% at 8 cores). We believe that a distributed protocol will not provide a faster futex implementation than the one that is in place. Indeed an alternative futex design, including modifications to *glibc*, should be introduced to Popcorn. Futex as it currently exists does not scale to a replicated-kernel design, as it is designed specifically as an optimization for an SMP OS.

## VIII. Conclusion

This work contributes the design of thread migration in a replicated-kernel OS, its implementation in Popcorn Linux, and an evaluation on selected benchmarks. The thread migration mechanisms investigated include the initial thread migration, and the maintenance of address space consistency. We show that the same design principles applied to the multikernel OS [2] can be applied to a monolithic OS adapted to the replicated-kernel OS model, while achieving similar scalability benefits. To make a multiple-kernel system run as Linux we extended the same system interface introducing inter-kernel migration instead of remote thread creation. Popcorn is shown to be faster then Barrelfish in most benchmarks and




(a) NPB IS (b) NPB FT (c) NPB CG (d) Rodinia BFS (e) Parboild LUD

Fig. 6. Execution time improvement on Popcorn with 4 and 8 prefetch slots over 1 slot varying the number of threads.

comparable to or better than Linux. In fact, Linux is up to 68% slower than Popcorn in the IS experiment. However, our study shows that the messaging layer acts as a primary bottleneck in the Popcorn implementation, e.g., slowing down thread migration, and considerably dampers scalability. We believe that hardware message passing will yield a more favorable scalability outcome. It was also found that scalability is dependent on application memory access patterns. Thus, we conclude that there may not be a catch-all solution that results in universally improved performance in a replicated-kernel OS design. Rather, application specific or dynamically determined policies may need to be developed to optimize performance for distributed actions.

## ACKNOWLEDGMENT

The authors thank the anonymous reviewers, Rauchfuss Holm, and Simon Peter for their valuable comments, and the Barrelfish team for their support. Popcorn Linux's messaging layer was originally written by Ben Shelton.

## REFERENCES

[1] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," ser. OSDI'08, 2008.

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

[3] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, 2009.

[4] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache coherence for gpu architectures," *Micro, IEEE*, vol. 34, no. 3, May 2014.

[5] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, "Atac: A 1000-core cache-coherent processor with on-chip optical network," ser. PACT '10, 2010.

[6] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Power challenges may end the multicore era," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.

[7] Tilera Corporation, http://www.tilera.com/products/platforms.

[8] Intel Corporation, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[9] D. A. Holland and M. I. Seltzer, "Multicore OSes: Looking forward from 1991, Er, 2011," ser. HotOS'13, 2011.

[10] R. W. Wisniewski, D. da Silva, M. Auslander, O. Krieger, M. Ostrowski, and B. Rosenburg, "K42: lessons for the OS community," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 1, Jan. 2008.

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

[12] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," ser. OLS '12, July 2012.

[13] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using rcu balanced trees," ser. ASPLOS XVII, 2012.

[14] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks summary and preliminary results," in *Supercomputing '91*, 1991.

[15] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from bell labs," in *UKUUG Conference*, 1990, pp. 1–9.

[16] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, Nov. 1989.

[17] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee, "Towards an efficient single system image cluster operating system," ser. 5th AAPP, Oct 2002.

[18] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, "A case for scaling applications to many-core with os clustering," ser. EuroSys '11, 2011.

[19] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: fault containment for shared-memory multiprocessors," ser. SOSP '95, 1995.

[20] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The sprite network operating system," *Computer*, vol. 21, no. 2, pp. 23–36, Feb 1988.

[21] S. McClure and R. Wheeler, "MOSIX: How linux clusters solve real world problems," ser. ATC '00, 2000.

[22] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using continuations to implement thread management and communication in operating systems," ser. SOSP '91, 1991.

[23] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," ser. OLS '02, 2002.

[24] A. Barbalace, B. Ravindran, and D. Katz, "Popcorn: a replicated-kernel os based on linux," ser. OLS '14, 2014.

[25] D. Katz, "Popcorn Linux: Cross Kernel Process and Thread Migration in a Linux-Based Multikernel," Virginia Tech, Tech. Rep., Sept. 2014.

[26] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," ser. SOSP '13, 2013.

[27] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep., Mar. 2012.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," ser. IISWC 2009, Oct 2009.