A Page Coherency Protocol for Popcorn Replicated-kernel Operating System

Marina Sadini
ECE, Virginia Tech
Blacksburg, VA, US
sadini@vt.edu

Antonio Barbalace
ECE, Virginia Tech
Blacksburg, VA, US
antoniob@vt.edu

Binoy Ravindran
ECE, Virginia Tech
Blacksburg, VA, US
binoy@vt.edu

Francesco Quaglia
DIAG, Sapienza
University of Rome, Italy
quaglia@dis.uniroma1.it

ABSTRACT

Popcorn is a Linux based replicated-kernel Operating System (OS). Popcorn was conceived as a research OS for a wide class of future heterogeneous-ISA hardware. Because of the novelty of such hardware, in which diverse OS-capable CPUs are glued together, it is not clear what level of memory sharing will be provided across these CPUs. In this paper we consider a setup in which diverse CPUs do not share memory. We addressed the problem of providing a coherent replicated process address space amongst different kernels, running on those CPUs, by proposing a new page coherency protocol. We deploy this protocol in Popcorn OS while adding additional functionalities to the Linux kernel to support inter-kernel thread migration and coordination. We tested and evaluated a prototype version of this work in an emulated environment; results show that the proposed page coherency protocol is effective and the implementation adheres to the model.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing/multiprogramming/multitasking, Threads*; D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms

Algorithms, Experimentation

Keywords

Software Page Coherency, Replicated-kernel, Operating System, Thread Migration, Linux

1. INTRODUCTION

Given the emerging trends in the computer community to increase parallelism and integrate ISA-diverse cores onto the same platform, and more recently onto the same chip (i.e. AMD Fusion [3], Intel Sandy Bridge [19]), we foresee the integration of diverse OS-capable cores on-die. Different OS-capable, overlapping-ISA, cores integrated on the same chip already exist: the ARM big.LITTLE architecture [17] integrates both power hungry (Cortex-A15) and power efficient (Cortex-A7) processors onto a single die. The goal of the big.LITTLE architecture is to provide a power efficient platform while maintaining high performance. The reasons to integrate ISA-heterogeneous cores onto the same platform or chip are numerous: load balancing, power efficiency, memory and device locality, faster execution, expanded memory space, greater device count, etc. Furthermore, although speculative, GPUs may become more OS-capable in the future (for example, the compute units of the Cell processor [8] are able to run a micro kernel).

Popcorn is a research OS based on the replicated-kernel model. In this model the OS consists of different kernel instances that are running in parallel on the same hardware; kernel instances communicate and cooperatively provide the illusion of a single system image (SSI) [9] to the running applications. The replicated-kernel model is the perfect fit in an ISA-heterogeneous setup; each kernel is compiled for an ISA-specific processor but architecture-agnostic communication mechanisms are used to allow the kernels to provide a single operating environment to the application (despite the ISA difference). Applications can be migrated on each processor in the system exploiting the unique features of each ISA (power efficiency, local peripherals, SIMD instructions, etc.). We currently do not address cross-ISA execution issues; we refer to the work of M. DeVujist et al. [5] instead. Popcorn OS is based on Linux, and is publicly available at `www.popcornlinux.org`.

1.1 Supporting Emerging Platforms

While the industry focus in recent years was to increase the number of computational units per chip, scientists became skeptical that cache coherency protocols would scale [1]. Recently, however, researchers have argued that cache coherency will be applicable to future many-core architectures [11]. Because of these contradictory trends, the structure of the memory architecture of future heterogeneous-ISA platforms is not clear. In a heterogeneous-ISA platform different cores can share memory, which can be hardware cache-coherent (e.g., big.LITTLE) or non-cache-coherent (e.g., SCC [20]); alternatively, cores may not share any memory (e.g., an Intel x86 server motherboard with a PCIe-connected Xeon Phi [7]). Platform inter- and intra- chip hardware messag-

ing can be adopted as a form of communication instead of shared memory, as previously envisioned [1].

Popcorn's goal is to transparently run applications, traditionally designed for SMP systems, on ISA-heterogeneous platforms, despite the underlying memory architecture. Due to the fact that a vast majority of applications were implemented by adhering to the SMP model, they do not have to be rewritten by using another programming paradigm, e.g. MPI or PGAS, when ported to a new platform where Popcorn is running. Mechanisms to support different memory architectures must be provided in Popcorn OS. In [14] we proposed sharing physical pages between kernels in a cache coherent shared memory setup, while coherently updating the virtual to physical mappings across kernels. We also proposed sharing the physical pages by adding an ownership rule and a cache flushing semantic in non-cache coherent shared memory configurations. In non-shared memory setups we propose to implement page replication per kernel.

1.2 Contributions

Although there are no fully heterogeneous-ISA platforms with shared memory available today (exceptions exist in the embedded market [18], although they have limitations), non-shared memory platforms can easily be built with commodity hardware. Several non-shared memory systems can be built from an x86 server by plugging PCIe-connected accelerators, intelligent network interface cards, disk controllers, etc. (e.g. Intel Xeon Phi, Tilera TILEncore, etc.). In this article, we present a new page coherency protocol which addresses the problem of how to keep different replicas of a process' address space coherent amongst different kernels running on processors that do not share memory. Our address space replication protocol addresses coherency at page granularity while considering the problem of replication at the virtual memory area (VMA) level (a group of pages with similar protection) to reduce the number of the pages that have to be kept coherent (e.g., read only areas can be loaded locally by each kernel). The novelty of the proposed protocol, compared to the state of the art, including MSI [13], IVY [10] and its optimizations in vNUMA [4], lies in its distributed design, instead of a centralized one. We believe this is a better fit for the hardware we are targeting.

We design and implement this protocol in Popcorn OS. Despite Popcorn running on real hardware, we tested and conducted an initial evaluation of the protocol in a multicore x86 emulation environment (QEMU). In addition to the algorithmic and technical details of the proposed protocol, in this paper we discuss the problems we faced while implementing it in the Linux memory subsystem. Furthermore we present the functionalities that were added to the Linux kernel in order to support thread migration, including a new signal, `SIGFORK`.

The article is organized as follows: Section 2 introduces related work, and in Section 3 we give a brief overview of Popcorn OS while describing the messaging subsystem and the thread migration mechanism. In Section 4 we present our page coherency protocol while in Section 5 implementation details are given. We present an initial evaluation in Section 6 and we conclude in Section 7.

2. RELATED WORK

A large body of work exists in the literature about hardware implementations of protocols to maintain cache coherency in multiprocessor environments. These protocols, including MSI, MESI, MOESI, have a long history [16] and are still evolving in order to scale to thousands of processors [11]. Our work addresses the problem of maintaining coherent process' address space replicas amongst different kernels that do not share memory within a software implementation.

K. Li and P. Hudak in [10] summarize their work on IVY, a distributed shared memory (DSM) setup; similarly to our setup they consider a parallel program as a set of threads. However, their work was developed for a network of workstations. Because of this, they made different implementation choices that we believe can be optimized for our environment (see Section 4) which inspired us to design a new algorithm. B. Fleisch and G. Popek in [6] implemented their DSM system in the OS kernel by using the architecture's page granularity. Similarly, our protocol is implemented in the OS kernel and maintains coherency at the page level.

M. Chapman and G. Heiser in [4] describe vNUMA, in which they propose several optimizations over the IVY protocol in order to build a virtual machine that spans different homogeneous-ISA SMP machines. Their work is different from ours in that we are providing transparency at the application level and not to the OS itself; additionally their approach cannot scale to heterogeneous-ISA hardware. A. Lebre et al. in [9] uses a variant of IVY to maintain a coherent distributed object in their kDDM module at the base of Kerrighed [9] (a Linux based cluster environment). Again this cannot scale to heterogeneous platforms. Furthermore, Popcorn is similar to single system image (SSI) extensions to operating systems that are designed for cluster environments, like Kerrighed; however existent cluster extensions only provide process migration.

To the best of our knowledge, only Agora [2] and Mermaid [12] addressed the problem of DSM in heterogeneous-ISA setups. Their approach to the problem is at a higher software layer where memory is still shared in pages but each page can only contain a single type of data with a specific interface. We aim to provide a transparent DSM-like system to the application.

Barrelfish multikernel OS [1] is the most advanced replicated-kernel OS. It was developed as a solution for homogeneous and heterogeneous ISA platforms. By date, to the best of our knowledge, Barrelfish does not provide support for replicated process address space. Furthermore, despite their mention of thread migration in [1], we didn't find any implementation in their source code.

3. POPCORN REPLICATED-KERNEL OS

Popcorn is an OS made up of several Linux kernels. Each kernel runs on a single CPU core or on a group of cores with hardware cache-coherent shared memory. As mentioned before, Popcorn aims to work across multiple architectures; different ISA cores may share memory or may not share memory. The goal of Popcorn is to provide the illusion of a single operating system amongst kernels to the applications regardless of the architecture. This work addresses non shared memory configurations. Therefore, on a heterogeneous-ISA platform, Popcorn will not require shared memory applications to be rewritten for non shared memory because the operating system will transparently handle diversity in the hardware (refer to Figure 1). In the following subsections we introduce the basic building blocks that are necessary to make this happen: the messaging layer and thread migration. These must be paired with the page coherency protocol in order to emulate the shared memory model expected by applications.

3.1 Messaging Layer

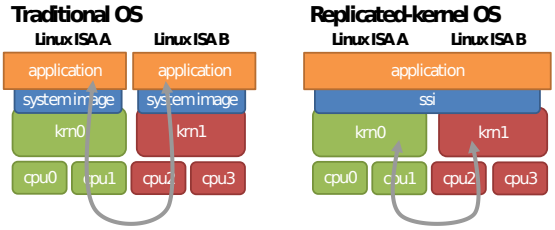


Figure 1: On a non-shared memory multicomputer, applications on different kernels must communicate explicitly (e.g., MPI) when a traditional OS is adopted. With a Replicated-kernel OS, SMP application can run without modifications instead.

In a platform in which ISA-diverse cores do not have access to any shared memory, a form of messaging must be exploited in order to make kernels communicate. In our Popcorn prototype which runs on multicore x86 hardware, we provide a software messaging layer on top of shared memory that simulates this setup. The messaging is built upon a combination of shared memory, buffering, and Inter Process Interrupts (IPI) to notify a remote kernel that a message has been delivered to it. Because of the not negligible dispatching latency of IPIs, we use a mixture of interrupts and polling in order to reduce the number of IPIs that are sent, thus increasing the overall throughput. Because of the fact that the multicore x86 AMD hardware, on which we are currently running Popcorn, does not support the x2APIC standard, we are not able to send IPIs in broadcast. Therefore, the messaging layer provides point-to-point communication only. This turns out to be a source of overheads which we considered in the design of the protocol (see Section 4).

3.2 Process and Thread Migration

In order to provide a process with the ability to exploit different hardware resources (like CPUs, peripherals, etc.) in parallel, on different cores, we developed inter-kernel process and thread migration in Popcorn. In the replicated-kernel design the computational model is different from the classical CPU-GPU, master-slave, approach. Because different OS-capable ISA-diverse cores will lie side-by-side on the same chip, we chose a computational model in which each core is a peer. This model extends the classic SMP multithreaded paradigm to heterogeneous-ISAs.

The focus of this article is on thread migration. Process migration amongst kernels does not require any address space coherency. For process migration, the entire process' address space is continuously migrated together with all threads making up the process; there is no replication, and hence no coherency is required. Thread migration per-se required a lot of engineering. We implemented thread migration atop the Popcorn process migration mechanism. Whenever several threads of a process migrate to a different kernel, on the kernel in which they are migrated, they must still be part of the same process (thread group in Linux). We implemented a mechanism that handles this scenario. When a thread is moved to a kernel it first searches for any other thread of the same thread group that was previously migrated to that kernel. If it finds a thread of the same thread group, it joins that thread group. When a process migrates to a different kernel, we create a new process, with parent `init`, that will mutate into the migrating process. In order to allow a migrating thread to join its thread group on a remote kernel we introduced a new signal, `SIGFORK`, that forces one of the threads of the thread group to clone itself and accommodate the new incoming thread.

Our process and thread migration mechanisms are running on homogeneous x86 multicore hardware, as we currently do not consider problems introduced by cross-ISA execution. Scheduling policies on heterogeneous-ISA platforms are out of the scope of this article.

4. PAGE COHERENCY PROTOCOL

We modelled our page coherency protocol over the MSI (Modified, Shared, Invalid) cache coherence protocol [13], Figure 2 shows our set of stable states. Similarly to hardware cache coherence protocols, our protocol guarantees strict coherence of the memory view to threads in a process. We maintain the same naming convention of stable states as in MSI, but a state is associated to a copy of a page (not a cache line) that belongs to a process virtual address space (instead of a chunk of physical memory). Furthermore we base our coherency protocol on the same *single-writer-multiple-reader* (SWMR) invariant [15], different other invariants are also considered.

A page in the Modified state allows the thread to perform reads and writes on the local version of the page. All the other copies in the system must be in the Invalid state. A page in the Shared state has an up-to-date version of the data; it can coexist in the system with other Shared or Invalid copies. In this state, thread reads can be performed on the local version of the page. A copy in the Invalid state is a non-valid version of the data, the up-to-date version has to be fetched from the system before any thread operation can continue.

Hardware cache coherency (snooping) protocols may rely on implicit ordering in accessing the bus to provide atomic operations across processors. Atomic message broadcasts may also be supported. Similarly, IVY assumes that all broadcast messages are atomic [10]. Hence an *Invalid* message atomically invalidates all the copies present in the system. Popcorn's messaging layer does not provide atomic operations or message broadcasts (see Section 3). We do not implement those functionalities in Popcorn but we create a coherency protocol that accommodates for their absence. This choice was motivated by the fact that a further software layer, between the messaging layer, and the coherency protocol, will increase the overhead in the application execution.

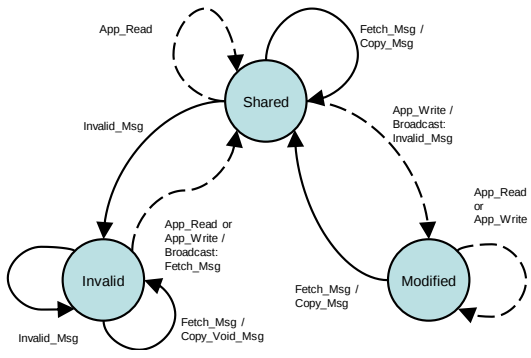


Figure 2: Protocol stable states (derived from MSI). Dashed lines represent transitions triggered by application code (memory accesses). Straight lines represent transitions triggered by remote messages.

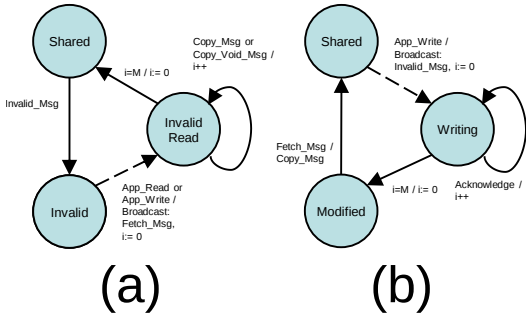


Figure 3: Left side Figure introduces the Invalid Read state. Right side Figure introduces the Writing state.

4.1 Wait States

Application threads running on Popcorn trigger read and write operations when pages are in stable states; such operations are intercepted and handled using our algorithm by the local kernel. Additional transient states are added to the stable ones to model inter-kernel communications. When a message is sent, it takes a not negligible amount of time before it is delivered to the receiver. The receiver may send back a response to the sender, which in turn will not happen instantaneously. The communication is modelled through wait states, in which the page copy should transit until it receives all the required answers. Application read and write could not be triggered on those states, because the data that the page copy holds is not valid. Therefore any read or write is delayed until the page copy reaches one of the stable states. The messages are queued on handlers in kernel space that are different from the ones that intercept the application's read or write operations (fault handlers). When all the answers related to a sent message reach the kernel, the page copy does not necessarily exit from the wait state immediately, but it could see a delay caused by the time needed to notify its handler.

4.2 Protocol Actions

Reads triggered on the Shared or the Modified states can be performed locally. When a read is triggered on the Invalid state instead, a *Fetch* message should be broadcast to the system. While waiting for the answers, the page copy transits in the Invalid Read state (refer to Figure 3). The other page copies in the system, upon receiving the *Fetch* message, should answer either with a *Copy_Void* message, i.e. such kernel does not have an updated copy of the data; or with a *Copy* message, that piggy-backs a valid copy of the data. An invariant of our algorithm is that at each time there either exists at least one page copy that will answer to the *Fetch* message, from a Shared state, or there exists a single page copy that will answer from a Modified state. This guarantees that at least one answer to the *Fetch* message contains an up-to-date copy of the data. After collecting all the answers, the page copy on which the read was performed can change its state to Shared. When a page copy in the Modified state answers to a *Fetch* with a *Copy* message, it changes its state to Shared.

Write operations can be performed locally only on kernels that own a page copy in Modified state. If the page is in an Invalid state, the most recent version of the data should be acquired before the write can occur. In this case we force a read action before the actual application write action. This implies that the state of the local version will change to Shared before the write will be processed. When a write occurs while the page is in Shared state, that kernel has to broadcast an *Invalid* message in the system. While waiting for all the answers, the page copy transits on a wait state, called Writing. To move from Writing to Modified state an acknowledge message from each kernel in the system caching a copy of that page must be received (refer to Figure 3). When a page copy in Shared state receives an *Invalid* message, its local data is no longer up-to-date so its state has to be changed to Invalid.

4.3 Resolving Concurrent Writes

In order to implement the single writer part of the SWMR invariant, at most one page copy in the Modified state can exist, at any time, in the system. If it exists it cannot coexist with Shared copies. When concurrent writes are triggered on more than one copy, only one of them should exit from the Writing state and reach the Modified one. The other copies should retry the write (*App_Write*).

In order to select which of those concurrent writers will reach the Modified state, we introduced a distributed voting scheme. When each page copy receives the *Invalid* message, it can vote (answer) with an *Ack* message or with a *Nack* message. Assuming that this distinction has been introduced to handle concurrent writes, only the copies in the Writing state will answer to an *Invalid* message with a *Nack*. Only the page copy that will receive all *Ack* messages will switch state to Modified, while the others have to perform the write again as they have an unsuitable copy.

In the Writing state time stamps are used to compute the answer to an *Invalid* message. When a write is triggered in the Shared state, the current local time is recorded during the transition to Writing. This time stamp will never change while in the Writing state, and it will be reset only when the page copy arrives in the Modified state. The *Invalid* message piggy-backs this time stamp. When an *Invalid* message is received, if the current state is Writing, the local time stamp is compared with the one in the message. If the time stamp in the message is greater than the local time stamp, then a *Nack* is sent. On the other hand, if these are equal we compare the kernel index. If the index of the receiving kernel is smaller than the index of the sender kernel, then a *Nack* is sent; otherwise an *Ack*. These two rules guarantee that only one kernel in the group of the concurrent writers will receive all *Ack* messages. Indeed only one will have the smaller time stamp if compared to the ones in the *Invalid* messages, otherwise if there are equal time stamps, just one will have the smaller index. In our target system there is no global clock, however each kernel has a monotonically increasing clock. This may imply that if a kernel starts a write at time $t1$ and another one starts a write in a time $t2 > t1$, the time stamp of the first kernel can be greater than the one of the second kernel. Because of that, the first kernel can be forced to retry the write. Once a write is repeated, the time stamp does not change. Due to the monotonically increasing clocks, this time stamp will become the smaller in the system eventually, and the page copy will reach the Modified state.

4.4 Expansion of the Writing State

To model the behavior described above, the Writing state is decomposed in three different sub-states: Writing Pend 1, Writing Pend 2 and Writing Read (see Figure 4). When a write is triggered on a Shared copy, the page copy transits in Writing Pend 1. If it receives all *Ack* messages, then it can

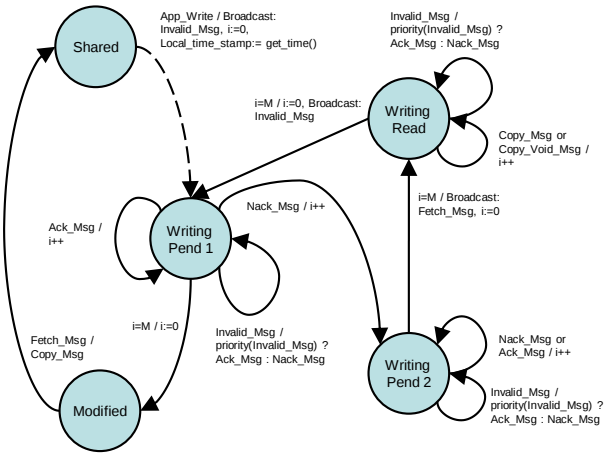


Figure 4: Expansion of the Writing state in Writing Pend 1, Writing Pend 2 and Writing Read.

switch state to Modified. On the other hand, if it receives a *Nack* it changes state to Writing Pend 2. The role of the state Writing Pend 2 is to collect all the remaining answers to the *Invalid*s previously sent. The page copies that are in Writing Pend 2 are the loser page copies, and when all the answers have arrived, they should fetch the new value of the data. They switch state to Writing Read and they broadcast a *Fetch* message. At least one page copy will answer with a *Copy* message, and when all the answers will be collected, the loser page copies will try to write again by broadcasting an *Invalid* message and changing state to Writing Pend 1. Page copies will loop in this cycle until their time stamp becomes the smaller in the system, and after a finite number of retries they will succeed.

4.5 Expansion of the Reading States

When a page copy is not trying to write and it receives an *Invalid* message, it has to answer to the sender with an acknowledge message and discard its local data because it is no longer up-to-date. Whenever such a page will be accessed again for a read, a valid copy must be gathered from the system by sending a *Fetch* message. An *Invalid* message during these states does not entail a loss of information on that data page.

However, if an *Invalid* message is received while in Invalid Read state, this possibly means that the collecting *Copy* messages piggy-back a value of the data that could have been invalidated by that same write in the sender kernels. The value that copies in Invalid Read states are going to replace can be potentially invalid. According to this scenario, when an *Invalid* is received in that state the *Fetch* message must be sent again, after all the answers have been received. To properly model this transition, Invalid Read is decomposed in two further sub-states: Invalid Read 1 and Invalid Read 2 (Figure 5). This can potentially lead to an infinite loop of read when the page copy is in Invalid Read. Indeed, in Invalid Read only *Ack* can be sent as an answer to an *Invalid* message. If iterated writes are performed on the other page copies, iterated *Invalid* messages could be received in the Invalid Read state before all the valid answers will reach the requestor.

4.6 Fetch and Fetch_Write Messages

The algorithm has to guarantee that for each *Fetch* message at least one page copy in the system will answer with

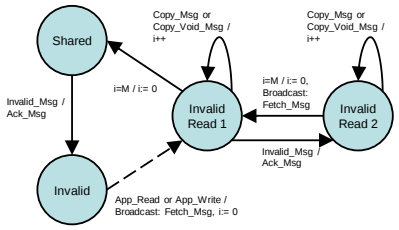


Figure 5: Expansion of Invalid Read state into Invalid Read 1 and Invalid Read 2.

a *Copy* message. It can happen that on all the page copies in the Shared state in the system, a write is triggered simultaneously. In this case, if a read is performed on another page copy in an Invalid state, these copies may answer with a *Copy_Void*, violating the invariant. Such a scenario may happen when the *Fetch* messages are received by the copies in the Writing state, before these will end up electing a writer. If a *Fetch* message is received in a Writing state, the answer must be delayed until the write is ended and the state becomes Modified. A delay in the answer guarantees that the elected page copy will answer with a *Copy* message when it arrives in the Modified state.

A non-elected page copy will retry to fetch an updated copy from the system. If the answers to these fetches are delayed while arriving in the Modified state, and if more than one concurrent page copy should retry the write, they will not be able to answer each other's respective fetches. In order to enable them to proceed, a new message has been introduced; to distinguish it from the *Fetch* message, we call it the *Fetch_Write* message. When fetching the data as a consequence of a write retry, *Fetch_Write* is sent. The semantic is the same of the *Fetch* message, but in the state Writing Read the answer is not delayed.

4.7 Concurrent Writes Oddities

Invalid Messages Handling. While electing a writer, not all concurrent page copies exit the electing phase at the same moment. When messages that are part of a previous election phase, reach a page copy after it has ended that phase, erroneous behaviors can manifest with the illustrated protocol.

When a page copy is in the Modified state no *Invalid* messages should be received because all the other page copies should be in the Invalid state, and should fetch the data before trying to write. However, when a page copy receives all the *Ack*s to its request to write, it does not mean that it has also received all the *Invalid* messages from the concurrent writers. Hence *Invalid* messages can be received after a page copy has switched to the Modified state. To handle this possibility we decided to answer to *Invalid* messages from a *Modified* state, but with *Nack* messages, that will be delivered to loser page copies.

Rounds. Another problem related to residual messages, arises when loser page copies try to fetch the most recent data content. The page copy in the Modified state is the only copy in the system that has the current valid data of a page. When all the concurrent page copies that lost the attempt to write are in the status Writing Read, they all expect a *Copy* message from the one in the Modified state. After the first *Fetch*, the page copy in Modified becomes Shared and starts to answer from that state. Before receiving all

the requests, however, one of the page copies, to which it has already answered, can change status to Writing Pend 1, and consequently send an *Invalid* message. The page copy in Modified state is obligated to change its state to Invalid before being able to diffuse the valid data content to all the concurrent page copies.

To solve this problem, a stricter form of coordination is introduced by defining rounds. Locally, rounds start when an *Invalid* message is sent. To know how many page copies are concurrently trying to write, when an *Ack* or *Nack* message is sent, a flag is set if the page copy is in a Writing states. When all the answers to an *Invalid* message are collected, the copy knows how many concurrent writers were in that round. If that page copy is the elected one, when in Modified state it will not answer to any *Fetch* or *Fetch_Write* message until all the concurrent page copies in the round, that elected it as the writer, have sent a request for data to it. This guarantees that the page copy in Modified state sends a *Copy* message to all the concurrent writers while they are in the Writing Read state. If the page copy is a loser one, instead, it has to strictly coordinate with the other non-elected page copies to advance its state within the round, in two steps. During the first step it waits in order to receive all the *Invalid* messages from the concurrent page copies, then it switches to Writing Read state. This step avoids the possibility of starting another retry round before the other copies will exit from the Writing Read state, causing all the answers to its request to become *Copy_Void* messages. In the second step, the page copy waits to receive all the *Fetch_Write* messages sent by the concurrent page copies, then it starts another round. Without this synchronization step a copy could start another round before having the chance to answer to the requests from concurrent page copies. That could cause starvation.

4.8 Adding Unmapped and Not Replicated States

When a page is utilized on more than one kernel, the page must be replicated on those kernels; the replicas that originate must be coordinated to provide a coherent memory view to the application. Once replicated, a page can be in any of the states presented so far, that all together make up the macro-state Replicated. It is likely that not all the kernels in the system are using a given page; the ones that are not using it do not need to start the replication algorithm. We introduce the Unmapped state to model the situation in which a page is not used by a kernel. All pages start from the Unmapped state. When only one kernel is using a page, and all the others have the page in the Unmapped state, the copy will be unique in the system, and there is no need for coordination. The page copy is in the Not Replicated state.

If a copy of a page is Replicated on one kernel, all the other page copies in the system must be in the Replicated state or Unmapped. At most one copy of a page can be in Not Replicated state in the system while all the other page copies have to be in Unmapped. When a kernel wants to access a page that is locally Unmapped, it has to broadcast a *Fetch* message to discover if some other kernel is using it. The page transits in a wait state, called Fetching, while it collects all the answers. When a kernel receives a *Fetch* message, if it has a page copy in Unmapped state, the answer will be *Copy_Void*. Instead, if it has a copy in Not Replicated state, the page is changing from unique in the system to become duplicated, so it changes the state to Shared and sends back a *Copy* message with the content of that page. If all the answers collected by the fetching kernel are *Copy_Void*, it means that it is the only one that is using that page, so it

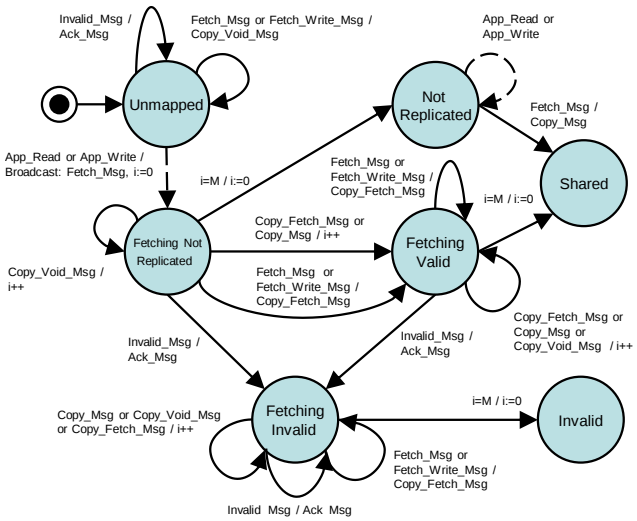


Figure 6: Unmapped and Not Replicated states subgraph.

can safely set the state to Not Replicated. If at least one *Copy* message is received, the page will become Replicated.

Nonetheless, two concurrent events can happen while fetching a page: an *Invalid* message can be received or another kernel is fetching concurrently a page that was Unmapped in the system. To model the behavior of our algorithm in these situations the Fetching state is decomposed into three substates: Fetching Not Replicated, Fetching Invalid and Fetching Valid (see Figure 6).

5. IMPLEMENTATION

The Linux kernel is a virtual memory (VM) OS. Applications and kernel code view virtual memory addresses. We exploit Linux's virtual memory subsystem to implement process' virtual address space replication.

The granularity of the physical to virtual mappings is the architecture page size; in x86 few page sizes are available, for our purposes we consider a unique page size, 4kB, while disabling huge pages. Virtual to physical translations are maintained in the page table. The page table format is architecture dependent, but generally, it has a tree data structure; leaf nodes are called Page Table Entry (PTE). A PTE contains the physical address of the page to which the virtual address, that we are translating, is referring to; furthermore it contains protection flags for that page. For each physical page in the system, Linux associates a `struct page` that mirrors such flags and maintains other vital information. Linux creates a page table for each user process and it implements on-demand paging. If a set of virtual addresses belonging to a page, are never accessed, the page is not mapped; i.e. no virtual to physical translation is available. Hence the MMU triggers a page fault exception on such addresses and Linux will add the translation for that page in the page table. The Linux kernel maintains a layout of the virtual address space of each process. Such layout provides enough information to the kernel itself in order to fill up the page table when the MMU triggers a page fault. This layout is maintained in a double linked list of `struct vm_area` (VMA). Each VMA refers to a virtual continuous set of pages with the same protection flags and type of backing storage (anonymous or file).

In the prototype version of Popcorn, developed for ccNUMA hardware, a process has the same virtual to physical memory map on all kernels, i.e. two threads of the same process running on different kernels have the same virtual address space and refer to the same physical pages. While adding page replication per-kernel, we modify this mechanism: two different threads of the same process have still the same virtual address space but they refer to different physical pages.

5.1 States Representation

The MMU on the x86 architecture generates an exception when it is not able to find a translation for a certain address, and whenever the access protections for a page are violated. PTE's flags allow the kernel to protect a page against writing (RW) and execution. To protect a page from being read there are two ways: remove it from the page table or clear the Present flag in the associated PTE.

We added a status field to the `struct page` that will reflect the replication state for a page. In the Linux page fault handler (`do_page_fault`) the `struct page` is accessed and the correct action is taken according to its state. To force the system not to access a page in an Invalid state we clear the Present bit in the PTE (such bit is normally used by the swap subsystem). Pages in the Shared and Modified states have the Present bit set in the PTE. If a page is in the Shared state the RW bit must be cleared: every write attempt will be caught by the MMU and run by our replication protocol.

5.2 Optimizations

When a process is executing on only one kernel, the replication protocol is off. Page faults are processed by the normal Linux virtual memory handler. When a process spans different kernels, likely only a subset of the address space is being used in parallel by its threads. The pages that belong to this subset are kept coherent by the replication algorithm. The subset of replicated pages is dynamically varying during execution. We apply on-demand paging instead of migrating the whole address space content in bulk with the thread. To improve the performance of the algorithm several fine tunings have been adopted.

Local fetch of read-only areas. Pages that belong to a read only VMA are not kept coherent by the replication protocol because no replica can modify their data during the application execution. Read only pages may be fetched from the local file system to reduce the communication overhead.

Multicast groups. A bitmap containing the indexes of kernels that are currently using a page is added to the `struct page`. This bitmap is updated each time a kernel receives a *Fetch* request from another kernel, and is piggy backed on the answer. This allows all kernels to keep an up-to-date version of the actual users of a page while reducing the amount of communication. Broadcast messages in the Replicated macro-state are substituted with multicasts to the subset of page's users.

Owner. An owner field is also added to the `struct page`. When a page is in the Replicated macro-state, but doesn't have an up-to-date version of the data, a multicast with a *Fetch* or *Fetch_Write* message should be sent. Likely, the kernel who sent the last *Invalid* message is the one whose write will succeed. Each time that an *Invalid* message is received, the owner of the page is set to the sender id if an *Ack* is sent back to it. By keeping track of the owner we can directly issue a fetch type of message to it instead of flooding a system with a multicast. Note that the owner can be changed while the fetch is happening; in that case the copy will send another fetch type message in multicast.

6. EVALUATION

In order to test the "correctness" of the page coherency protocol we developed different (memory bound) micro benchmarks. These consist of multithreaded programs that concurrently and iteratively read and write a shared data structure. We verified that the protocol provides a strict coherent virtual address space view, on different kernels, by running several parallel computational algorithms, with different degrees of synchronization and sharing (FFT, CG, IS, etc.). We compared the output of those computational programs when running on Popcorn and on vanilla Linux with the same input; our tests show that the outputs always matched, hence showing the effectiveness of the protocol.

In the following we present an initial evaluation of the protocol, implemented in Popcorn. We run the experiments in QEMU x86 while loading a setup with Popcorn OS running over two kernels. The test application launches two threads, each pinned on a different kernel; the first thread allocates an array then creates the second thread that migrates on the other kernel. Threads synchronize on a shared variable and then start to operate concurrently on the array in a sequential fashion for a different number of iterations (1, 10, 30, 60, 90). We recorded the number of messages exchanged in the system to keep the process address space replicas coherent. We analyzed the following cases: both threads are reading the array (Figure 7(b)), one thread is writing and the other is reading (Figure 7(c)), both threads are writing (Figure 7(d)). We repeat each experiment with different array sizes (that fits in less than one page, in one page and in 64 pages).

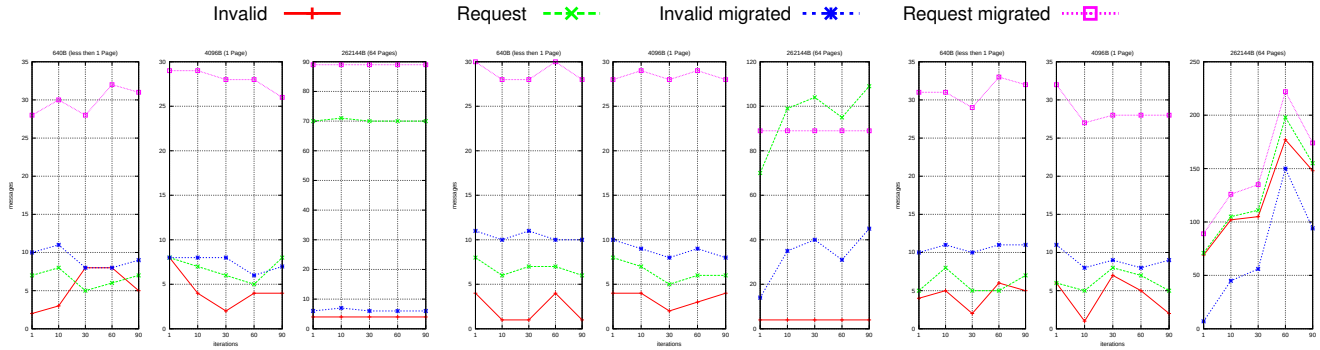
In Figure 7(b) we notice that, as expected for concurrent readers, increasing the number of iterations does not change the number of page requests (*Fetch* and *Fetch_Write* messages) in the system. All pages are locally copied during the first iteration; no more requests are generated in the following iterations (because no updates are made).

Figure 7(c) shows that in the case of one thread reading and the other writing, the number of messages to update the reader is greater then in the previous case. All the other messages are kept around the same amount, hence the communication overhead is not noticeably increased.

In Figure 7(d) we notice that writing on both threads generates more messages in the system due to the increased number of *Invalid* sent by both copies. This reflects the behavior of the proposed protocol hence insuring that the implementation adheres to the design. It is worth noting that in Figure 7(c) and Figure 7(d), for the case of 64 pages, the number of messages is low due to the implementation choices in the messaging layer (that reflects the behavior of the expected non-shared memory heterogeneous-ISA setup). Because locally accessing the entire array is comparable to the cost of sending a message, and per-message queues are independently scheduled by the kernel, a thread operating on the array may finish a few iterations without sending any message. Finally, consistently over all the three cases, changing the array size up to a page makes the protocol generate traffic for the same amount of messages; most of this traffic is generated due to thread management and coordination.

7. CONCLUSIONS

We presented a new page coherency protocol that guarantees a strict coherent view of a process address space when replicated on different kernels in a replicated-kernel OS.



(b) **read-read** application (c) **read-write** application (d) **write-write** application

Figure 7: *Request* and *Invalid* messages exchanged in the system.

We introduced our protocol by extending the MSI cache coherency protocol with other stable and transient states. We incrementally defined a set of invariants, which we built on to define the protocol (in terms of new states, messages or actions). We demonstrate that the invariants hold. This work turns into a new protocol specification, innovative from previous works, including MSI, IVY and vNUMA.

A preliminary implementation of the protocol has been integrated in Popcorn OS. New software components have been added to the Popcorn and Linux source codes to support per-group-of-threads replicated address space. Furthermore, we deployed different optimizations to reduce kernel to kernel communication overhead due to the coherency protocol.

We evaluate the implementation over different micro benchmarks in an emulated environment, while showing that the design properties are satisfied and the implementation is effective.

In the future we plan to test and compare the replication protocol on real hardware, including multicore x86, and non-shared memory PCIe-connected heterogeneous-ISA combinations. We are currently working on the cross-ISA messaging layer and cross-ISA execution migration. Furthermore, we will explore how to exploit the different memory consistency models, deployed by cpu architectures, to improve scalability of the protocol.

8. REFERENCES

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, pages 29–44. ACM, 2009.

[2] R. Bisiani, C.-M. U. C. S. Dept, F. Alleva, F. Correrini, F. Lecouat, and R. Lerner. *Heterogeneous Parallel Processing: the Agora Shared Memory*. CMU-CS. 1987.

[3] P. Boudier and G. Sellers. Memory system on fusion APUs: The benefits of zero copy. Technical report, 2011.

[4] M. Chapman and G. Heiser. vnuma: a virtual shared-memory multiprocessor. ATC '09, pages 2–2, Berkeley, CA, USA, 2009. USENIX Association.

[5] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. ASPLOS XVII, pages 261–272. ACM, 2012.

[6] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. *SIGOPS Operating Systems Review*, 23(5):211–223, Nov. 1989.

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

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[9] A. Lèbre, R. Lottiaux, E. Focht, and C. Morin. Reducing kernel development complexity in distributed environments. Euro-Par '08, pages 576–586, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[11] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, jul 2012.

[12] I. national de recherche en informatique et en automatique (France). *Proceedings: The 10th International Conference on Distributed Computing Systems, Paris, France, May 28-June 1, 1990.* Number v. 10. Ieee Computer Society Press, 1990.

[13] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface.* Morgan Kaufmann, 2005.

[14] M. Sadini, D. Katz, A. Barbalace, A. Murray, and B. Ravindran. Towards replicated-kernel os support for task migration on heterogeneous-isa platforms. SOSP '13. ACM, 2013.

[15] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011.

[16] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[17] A. Stevens. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. Technical report, 2011.

[18] Texas Instruments Inc. OMAP4460 multimedia device technical reference manual. Technical report.

[19] B. Valentine. Introducing sandy bridge. Technical report.

[20] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on Intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, Feb. 2011.