Popcorn Linux: Cross Kernel Process and Thread Migration in a Linux-Based Multikernel

David G. Katz

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Robert Broadwater
Chao Wang

September 1, 2014
Blacksburg, Virginia

Keywords: Linux, Multikernel, Process, Thread, Address Space Migration, Task Migration

Popcorn Linux: Cross Kernel Process and Thread Migration in a Linux-Based Multikernel

David G. Katz

(ABSTRACT)

Proliferation of new computing hardware platforms that support increasing numbers of cores, as well as increasing instruction set architecture (ISA) heterogeneity, is creating opportunity for systems software developers to question existing software architecture.

One promising emerging systems architecture is the *multikernel*, pioneered in Barrelfish OS. The multikernel directly addresses the challenges of high core counts and increased heterogeneity by partitioning the system into multiple independently running kernel instances which cooperate to form a single operating system. Popcorn Linux is an adaptation of the multikernel concept to a Linux environment, melding the multikernel concept with the power and ubiquity of the Linux platform. The goal of the Popcorn Linux project is to provide a Linux-based single-system image environment for heterogeneous hardware. In constructing this environment, Linux must be extended to distribute the plethora of operating system services that it provides across kernel instances.

This thesis presents the newly developed Popcorn Linux mechanism for migrating tasks and their address spaces between kernel instances at arbitrary points in their execution. Both process and thread migration is supported, and distributed address spaces are maintained and guaranteed to remain consistent between distributed thread group members running on different kernel instances. Tasks can migrate through an unlimited number of kernel instances, as well as back to previously visited kernel instances. Additionally, the full task life-cycle is supported, allowing migrated tasks to exit and create new children on whichever kernel instance happens to be hosting them.

The mechanisms developed were vetted through unit testing, review, and a number of compute-bound benchmarks in a homogeneous x86 64bit environment. Correctness was demonstrated, and performance metrics were acquired. Popcorn Linux performance was shown to be reasonable when compared to SMP Linux. The mechanisms developed are therefore deemed feasible. Scalability was determined to be a function of workload characteristics, where in some cases Popcorn Linux out-scales SMP Linux and in other cases SMP Linux out-scales Popcorn Linux. Optimizations are recommended to reduce the maturity gap between Popcorn Linux and SMP Linux, improving Popcorn Linux performance.

Dedication

This work is dedicated to my wife and my daughter.

Contents

1 Introduction **1**

1.1 Research Contributions 4

1.2 Scope 4

1.3 Thesis Organization 5

2 Related Work **6**

2.1 Multikernels 6

2.1.1 Barrelfish Operating System 6

2.1.2 Factored Operating System (FOS) 7

2.2 Cluster Operating Systems 7

2.2.1 Disco 7

2.2.2 Hive 8

2.3 Linux Approaches to Multikernel Design 9

2.3.1 NoHype 9

2.3.2 coLinux 10

2.4 Software Partitioning of Hardware 11

2.4.1 Twin Linux 11

2.4.2 SHIMOS 12

2.5 Compute-Node Kernels 12

2.6 Operating Systems for Emerging Exascale Multicore 14

2.6.1 Tessellation 14

2.6.2 Tornado . . . 14
2.6.3 Corey . . . 15
2.6.4 Sprite . . . 16
2.7 Scalable Virtual Memory Subsystems . . . 17
2.7.1 Bonsai . . . 17
2.7.2 RadixVM . . . 17
2.8 Limitations of Previous Work . . . 18
2.8.1 Single system image . . . 18
2.8.2 Address space consistency . . . 18

3 Linux Concepts **19**
3.1 Linux Execution Contexts . . . 19
3.2 Task Creation . . . 20
3.3 Task Exiting . . . 21
3.4 Scheduling . . . 22
3.5 Linux Memory Management . . . 22
3.6 Linux User Mode Helper API . . . 24

4 Popcorn Concepts **25**
4.1 Popcorn System Architecture . . . 25
4.2 Hardware Partitioning . . . 25
4.3 Global Accessible Memory . . . 26
4.4 Message Passing . . . 26
4.5 Device Drivers . . . 26
4.6 Single System Image . . . 27
4.7 Applications Spanning Kernel Instances . . . 27
4.8 Basic Building Blocks . . . 27

5 Task Migration **29**
5.1 Architectural Overview . . . 29

5.2 Shadow Tasks . . . 30
5.3 Distributed Task and Thread Group Identification . . . 31
5.4 Forking Children . . . 31
5.5 Task Exit Procedure . . . 33
5.6 Task Migration Mechanism . . . 34
5.6.1 Migrating A Task To A New Kernel Instance . . . 35
5.6.2 Migrating A Task To A Previous Kernel Instance . . . 36
5.7 Address Space Migration and Consistency Maintenance . . . 36
5.7.1 Address Space Consistency . . . 37
5.7.2 Up-Front Address Space Migration . . . 40
5.7.3 On-Demand Address Space Migration . . . 41
5.7.4 Concurrent Mapping Retrieval . . . 44
5.7.5 Virtual Memory Area Modification . . . 45
5.7.6 Mapping Prefetch . . . 51

6 Results **53**
6.1 Microbenchmarks . . . 54
6.1.1 Mechanism Costs . . . 54
6.1.2 Task Migration Measurement . . . 55
6.2 IS-POMP . . . 56
6.2.1 IS-POMP Workload Profile . . . 56
6.2.2 IS-POMP Results . . . 66
6.3 FT-POMP . . . 68
6.3.1 FT-POMP Workload Profile . . . 68
6.3.2 FT-POMP Results . . . 77
6.4 CG-POMP . . . 80
6.4.1 CG-POMP Workload Profile . . . 80
6.4.2 CG-POMP Results . . . 89
6.5 BFS-POMP . . . 92

6.5.1 BFS-POMP Workload Profile . . . 92
6.5.2 BFS-POMP Results . . . 101
6.6 LUD-POMP . . . 103
6.6.1 LUD-POMP Workload Profile . . . 104
6.6.2 LUD-POMP Results . . . 112
6.7 Benchmark Discussion . . . 115
6.7.1 SMP Linux versus Popcorn Linux . . . 115
6.7.2 Mapping Retrieval Overhead and Symmetry . . . 115
6.7.3 Prefetch . . . 116
6.7.4 Transport Speed . . . 117

7 Conclusions **118**

8 Future Work **120**
8.1 Identify and Resolve Performance Bottlenecks . . . 120
8.2 Alternative Mapping Retrieval Protocols . . . 120
8.2.1 Single Query Single Response Mapping Query . . . 121
8.2.2 Distributed Thread Grouping Based Mapping Query . . . 122
8.3 Shared File System . . . 123
8.4 User-Space Locking Abstractions . . . 123
8.5 Scheduling Support . . . 123

List of Figures

1.1 Perf Measurement for IS-POMP Workload on SMP Linux 2

4.1 Popcorn System Overview 25

5.1 Migration - originating kernel instance 30

5.2 Migration - receiving kernel instance 31

5.3 Task duplication hook 32

5.4 Task exit procedure 33

5.5 Response to thread group exiting notification 34

5.6 Two consistent distributed VMAs in a distributed address space. In light color the full VMA address range, in solid color the VMA mappings known by each kernel instance. 38

5.7 An inconsistent distributed VMA in a distributed address space. In light color the full VMA address range, in solid color the VMA mappings known by each kernel instance. 39

5.8 On-demand address space migration procedure 42

5.9 Munmap procedure 47

5.10 Mprotect procedure 48

5.11 Mremap procedure 49

5.12 Mmap procedure 50

6.1 Migration time measurements comparing SMP to Popcorn Linux 55

6.2 IS Workload Event Totals With 1 Prefetch Slot 57

6.3 IS Workload profile with 16 kernel instances with 1 prefetch slot 58

6.4 IS Workload Event Totals With 4 Prefetch Slots 59
6.5 IS Workload profile with 16 kernel instances with 4 prefetch slots 60
6.6 IS Workload Event Totals With 8 Prefetch Slots 61
6.7 IS Workload profile with 16 kernel instances with 8 prefetch slots 62
6.8 Perf Measurement for IS-POMP Workload on SMP Linux 64
6.9 Perf Measurement for IS-POMP Workload on SMP Linux - User vs Kernel . 64
6.10 Perf Measurement for IS-POMP Workload on Popcorn 65
6.11 Perf Measurement for IS-POMP Workload on Popcorn - User vs Kernel . . . 65
6.12 is-pomp results varying involved kernel instances with 1 prefetch slot 66
6.13 is-pomp results varying involved kernel instances with 4 prefetch slots 67
6.14 is-pomp results varying involved kernel instances with 8 prefetch slot 67
6.15 is-pomp results varying kernel instances involved 68
6.16 FT Workload Event Totals With 1 Prefetch Slot 69
6.17 FT Workload profile with 16 kernel instances with 1 prefetch slot 70
6.18 FT Workload Event Totals With 4 Prefetch Slots 72
6.19 FT Workload profile with 16 kernel instances with 4 prefetch slots 72
6.20 FT Workload Event Totals With 8 Prefetch Slots 73
6.21 FT Workload profile with 16 kernel instances with 8 prefetch slots 74
6.22 Perf Measurement for FT-POMP Workload on SMP Linux 75
6.23 Perf Measurement for FT-POMP Workload on SMP Linux - User vs Kernel 76
6.24 Perf Measurement for FT-POMP Workload on Popcorn 76
6.25 Perf Measurement for FT-POMP Workload on Popcorn - User vs Kernel . . 77
6.26 ft-pomp results varying involved kernel instances with 1 prefetch slot 78
6.27 ft-pomp results varying involved kernel instances with 4 prefetch slots 78
6.28 ft-pomp results varying involved kernel instances with 8 prefetch slot 79
6.29 ft-pomp results varying kernel instances involved 79
6.30 CG Workload Event Totals With 1 Prefetch Slot 81
6.31 CG Workload profile with 16 kernel instances with 1 prefetch slot 82

6.32 CG Workload Event Totals With 4 Prefetch Slots 83
6.33 CG Workload profile with 16 kernel instances with 4 prefetch slots 84
6.34 CG Workload Event Totals With 8 Prefetch Slots 85
6.35 CG Workload profile with 16 kernel instances with 8 prefetch slots 86
6.36 Perf Measurement for CG-POMP Workload on SMP Linux 87
6.37 Perf Measurement for CG-POMP Workload on SMP Linux - User vs Kernel 88
6.38 Perf Measurement for CG-POMP Workload on Popcorn 88
6.39 Perf Measurement for CG-POMP Workload on Popcorn - User vs Kernel . . 89
6.40 cg-pomp results varying involved kernel instances with 1 prefetch slot 90
6.41 cg-pomp results varying involved kernel instances with 4 prefetch slots 90
6.42 cg-pomp results varying involved kernel instances with 8 prefetch slot 91
6.43 cg-pomp results varying kernel instances involved 91
6.44 BFS Workload Event Totals With 1 Prefetch Slot 93
6.45 BFS Workload profile with 16 kernel instances with 1 prefetch slot 94
6.46 BFS Workload Event Totals With 4 Prefetch Slots 95
6.47 BFS Workload profile with 16 kernel instances with 4 prefetch slots 95
6.48 BFS Workload Event Totals With 8 Prefetch Slots 97
6.49 BFS Workload profile with 16 kernel instances with 8 prefetch slots 97
6.50 Perf Measurement for BFS-POMP Workload on SMP Linux 99
6.51 Perf Measurement for BFS-POMP Workload on SMP Linux - User vs Kernel 99
6.52 Perf Measurement for BFS-POMP Workload on Popcorn 100
6.53 Perf Measurement for BFS-POMP Workload on Popcorn - User vs Kernel . . 100
6.54 bfs-pomp results varying involved kernel instances with 1 prefetch slot 101
6.55 bfs-pomp results varying involved kernel instances with 4 prefetch slots . . . 102
6.56 bfs-pomp results varying involved kernel instances with 8 prefetch slot 102
6.57 bfs-pomp results varying kernel instances involved 103
6.58 LUD Workload Event Totals With 1 Prefetch Slot 104
6.59 LUD Workload profile with 16 kernel instances with 1 prefetch slot 105

6.60 LUD Workload Event Totals With 4 Prefetch Slots 106
6.61 LUD Workload profile with 16 kernel instances with 4 prefetch slots 107
6.62 LUD Workload Event Totals With 8 Prefetch Slots 108
6.63 LUD Workload profile with 16 kernel instances with 8 prefetch slots 109
6.64 Perf Measurement for LUD-POMP Workload on SMP Linux 110
6.65 Perf Measurement for LUD-POMP Workload on SMP Linux - User vs Kernel 111
6.66 Perf Measurement for LUD-POMP Workload on Popcorn 111
6.67 Perf Measurement for LUD-POMP Workload on Popcorn - User vs Kernel . 112
6.68 lud-pomp results varying involved kernel instances with 1 prefetch slot . . . 113
6.69 lud-pomp results varying involved kernel instances with 4 prefetch slots . . . 113
6.70 lud-pomp results varying involved kernel instances with 8 prefetch slot . . . 114
6.71 lud-pomp results varying kernel instances involved 114

List of Tables

6.1 Microbenchmark results 54
6.2 IS-POMP fault processing times by kernel instance with 1 prefetch slot . . . 58
6.3 IS-POMP fault processing times by kernel instance with 4 prefetch slots . . . 61
6.4 IS-POMP fault processing times by kernel instance with 8 prefetch slots . . . 62
6.5 IS-POMP 2 Mapping Retrieval Message Transport Times 63
6.6 FT-POMP fault processing times by kernel instance with 1 prefetch slot . . 71
6.7 FT-POMP fault processing times by kernel instance with 4 prefetch slots . . 73
6.8 FT-POMP fault processing times by kernel instance with 8 prefetch slots . . 74
6.9 FT-POMP 2 Mapping Retrieval Message Transport Times 75
6.10 CG-POMP fault processing times by kernel instance with 1 prefetch slot . . 82
6.11 CG-POMP fault processing times by kernel instance with 4 prefetch slots . . 85
6.12 CG-POMP fault processing times by kernel instance with 8 prefetch slots . . 86
6.13 CG-POMP 2 Mapping Retrieval Message Transport Times 87
6.14 BFS-POMP fault processing times by kernel instance with 1 prefetch slot . . 94
6.15 BFS-POMP fault processing times by kernel instance with 4 prefetch slots . 96
6.16 BFS-POMP fault processing times by kernel instance with 8 prefetch slots . 98
6.17 BFS-POMP 2 Mapping Retrieval Message Transport Times 98
6.18 LUD-POMP fault processing times by kernel instance with 1 prefetch slot . . 105
6.19 LUD-POMP fault processing times by kernel instance with 4 prefetch slots . 108
6.20 LUD-POMP fault processing times by kernel instance with 8 prefetch slots . 109
6.21 LUD-POMP 2 Mapping Retrieval Message Transport Times 110

Chapter 1

Introduction

As core count increases in hardware platforms, researchers are questioning the scalability of the traditional and ubiquitous shared memory symmetric multiprocessing (SMP) operating system design. Some researchers have found that SMP systems can continue to scale[6]. Others argue that extending scalability optimizations made for a given architecture to different architectures often involves prohibitive amounts of work[3]. Yet others point out that an iterative process must be applied to eliminate bottlenecks in SMP systems, where each new advance reveals a new bottleneck[8]. Lock contention in SMP Linux was measured in this effort under various workloads, and shown to add significant amounts of overhead, specifically in the memory management sub system. Contention over memory management resources is expected in SMP systems due to the presence of multiple concurrently running CPUs experiencing faults on the same memory map, and as a result needing to modify page tables and other highly shared data structures. Figure 1.1 shows SMP overhead as seen on the main thread of a cpu-bound workload. As the number of kernel instances increases, lock contention seen in *handle_pte_fault* and *pagevec_lru_move_fn* increasingly dominates overhead. These functions are both part of the memory management sub system that handle fault events. In addition to these types of scalability issues, SMP systems suffer from high failure rates due to poor isolation between cores[8]. In response to these observations, new systems software strategies that stray from SMP tradition are emerging to efficiently harness the computing power that hardware platforms provide.

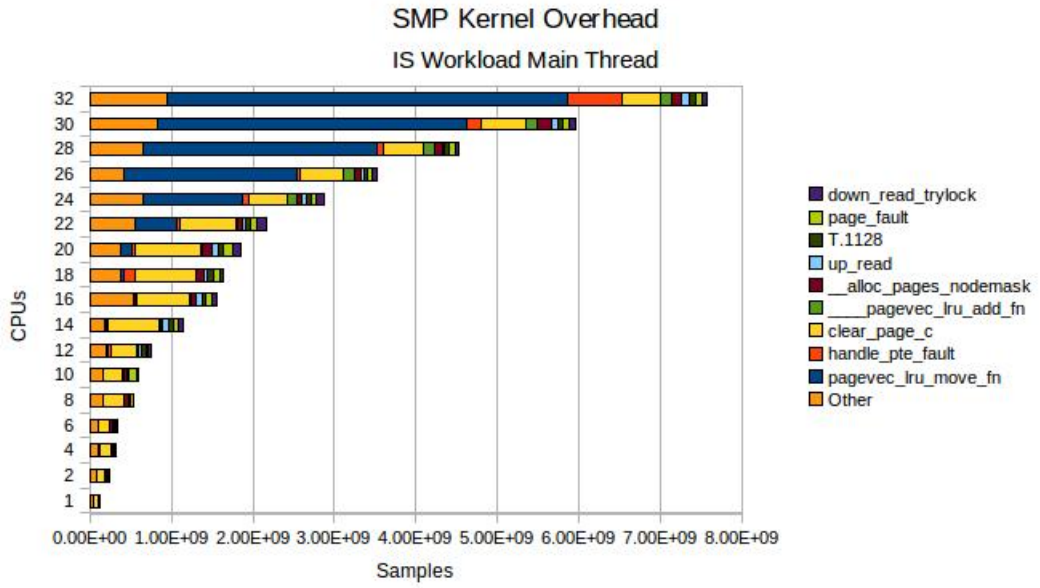

Figure 1.1: Perf Measurement for IS-POMP Workload on SMP Linux

Not only is core count increasing, ISA heterogeneity is also being introduced at an increasing rate in shared memory systems. The traditional shared memory SMP paradigm breaks down once ISA heterogeneity is introduced. Cores with different instruction sets cannot share code, and therefore sharing a monolithic kernel image is not possible. A solution to this problem that is seen in many current technologies is to run separate operating systems on cores of dissimilar types, as found in the Qualcomm MSM family of SoCs[13]. Multicore chipsets hosting cores of similar ISA but of varied types and capabilities are also emerging. These chipsets are seen in the mobile devices space (smart phones and tablets), where power concerns are significant. ARM big.LITTLE is one such emerging heterogeneous architecture characterized by some number of powerful ARM processors coexisting with a number of relatively less powerful ARM processors on the same die, sharing a common and coherent cache. The high power chips are used for CPU intensive tasks, while background processing is handled on the more power-frugal CPUs. This architecture has been used in a number of Samsung, Renesas Mobile, MediaTek, and Qualcomm ARM chips for use in mobile devices[14]. Heterogeneity is also found in current workstations. Intel integrates a Xeon processor with a Xeon Phi coprocessor cluster over a PCIe interconnect. The instruction sets of the Xeon and Xeon Phi devices are overlapping but not equal.

Many of the emerging systems concepts focus on identifying new methods of pairing computation with computing resources. One such concept is the *multikernel*. The concept of a multikernel, introduced by Bauman et al., of the Systems Group at ETH Zurich and Microsoft Research, is embodied in the Barrelfish project and is a design strategy that departs from the traditional SMP notion of how to structure an operating system in a multi/many-

core potentially heterogeneous architecture. This solution directly addresses scalability issues arising from increased core count, as well as the coupling issue that is inherent to heterogeneous architectures. The multikernel is a replicated-kernel operating system where each hardware resource, *e.g.* core, group of cores sharing a common ISA, non-uniform memory access (NUMA) node, or cluster thereof, runs an independent kernel instance out of its own private portion of the memory space. Each kernel instance acts as a peer, cooperating with the others to form a cohesive, single system image environment. In contrast to a traditional SMP operating system, kernel data structures are not shared between kernel instances. Instead, all kernel layer communication is done explicitly through message passing between kernel instances. This facilitates cooperation, while maintaining separation to allow kernel instances to run independently. This independence removes lock contention between kernel instances, as kernel data structures are replicated rather than being shared and therefore require no synchronization. This independence also facilitates heterogeneity within the multikernel by allowing kernel instances running on dissimilar ISAs to cooperate by using an agreed upon messaging interface.

The Popcorn Linux project is a research effort aimed at building scalable systems software for future generations of heterogeneous and homogeneous multi/many-core architectures. Popcorn Linux is a self-hosting multikernel operating system where each kernel instance is a modified Linux kernel. Many Popcorn Linux kernel instances co-exist on the same hardware and can be allocated to hardware resources one per core, one per ISA, one per NUMA node, or clustered in some combination thereof. Hardware resources, such as disks and network interfaces, can be assigned (and dynamically re-assigned) kernel instance affinity to give exclusive access to that resource to a given kernel instance.

All kernel instances in the multikernel must cooperate in order to stitch together a single logically consistent user space to host user layer software. In order to host legacy Linux applications, the system must handle this cooperation at the kernel layer, allowing the user layer software to implement business logic. This is a sizable task in a multikernel environment, as all resources and functionality that the user software relies upon must be carefully managed in a distributed fashion.

One requirement for creating this unified user space is effective task migration across kernel instances. This capability allows for workload distribution and balancing within the multikernel. This thesis is focused on task migration in a homogeneous x86 64bit multikernel environment, with attention given to both process and thread migration. The ability to migrate a task at any arbitrary point in its execution between kernel instances was developed, tested, and characterized.

Address space consistency across distributed thread group members is a significant problem in an environment where members of the thread group may be executing on different kernel instances. This thesis breaks that problem into components and presents viable solutions to each aspect of address space consistency, each of which has been vetted through implementation and testing.

While this thesis concentrates exclusively on task migration and address space consistency in a homogeneous x86 64bit environment, it can be used as a stepping stone to task migration in other homogeneous environments and eventually heterogeneous environments. The challenges identified in the area of task migration and address space consistency will continue to be problems in other architectures. As Popcorn Linux is developed into a heterogeneous systems environment, the solutions presented in this thesis can be extended to include the additional logic that is necessary to support cross architecture migrations.

1.1 Research Contributions

This thesis contributes the following to the development of the Popcorn Linux platform:

1. A mechanism is designed and implemented to migrate tasks between kernel instances in a homogeneous x86 64bit multikernel environment.
2. A mechanism is designed and implemented to migrate address space components as needed to support migrated tasks.
3. A mechanism is designed and implemented to ensure that distributed address spaces remain consistent across kernel instances.
4. The implementation of the components listed above are tested against five benchmark workloads, and the results that were collected are analyzed. Recommendations for refining the design are made based on insights gained from the analysis. The resulting system is shown to perform reasonably when compared to SMP Linux and is therefore a feasible approach. Specific improvements can be made to Popcorn Linux to realize further gains relative to SMP Linux.

1.2 Scope

This thesis does not touch upon resource migration, such as the migration of open file descriptors, locks, drivers, and name spaces including process identifier (PID) name space. Other students are working on those aspects of the single system image development. A mechanism for migrating file descriptors is also currently under development by SSRG, but is not part of this thesis. This thesis focuses exclusively on task and address space migration and consistency.

1.3 Thesis Organization

Chapter 2 provides information about related work. Special attention is given to task migration, address space migration, and address space indexing mechanisms when applicable.

Chapter 3 presents Linux concepts that are necessary to an understanding of Popcorn Linux.

Chapter 4 presents Popcorn Linux concepts including the overall system architecture.

Chapter 5 is an in-depth description of task migration in Popcorn Linux. This chapter also describes the methods used to achieve the distributed address spaces necessary to support distributed thread groups.

Chapter 6 presents the results achieved. This section profiles benchmark workloads to determine the types of operations the kernel is performing. Specific benchmarks that vary a number of kernel configuration items are then presented. Results are analyzed to identify trends.

Chapter 7 provides general concluding remarks.

Chapter 8 provides some suggestions for areas to further research.

Chapter 2

Related Work

This chapter provides a survey of existing projects that are related to multi-core systems software. To provide a backdrop for this thesis, this section focuses on task migration and address space indexing mechanisms.

2.1 Multikernels

2.1.1 Barrelfish Operating System

Systems Group at ETH Zurich, Microsoft Research, and ENS Cachan Bretagne introduce the notion of a multikernel [3]. Barrelfish, their implementation of the multikernel concept, is predicated on strict separation between operating system (OS) nodes. An OS node is mapped in a one to one relationship to a central processing unit (CPU) core. OS nodes do not communicate through shared OS internal data structures. Rather message passing is used explicitly for all communications between OS nodes using pages designated exclusively for messaging. This model uses state replication to create a unified user space via asynchronous message passing between OS nodes. The creators of Barrelfish suggest that the multikernel system can be architecture agnostic to support the increasingly heterogeneous nature of multi-core systems. The data structures used in the multikernel are hardware-neutral and can therefore be migrated between CPUs of disparate type. However, Barrelfish does not yet implement this functionality.

Processes in the Barrelfish environment are structured and migrated differently when compared to an SMP Linux environment. In Barrelfish, a process is represented in an OS node by a component called a *dispatcher*. There is one dispatcher per process on each OS node. When a task is migrated to a new OS node, the dispatcher for that process on the original OS node makes an up-call into a user- space thread scheduler which handles the migration. In contrast, the Linux SMP migration mechanic uses shared OS data structures in kernel

space to stop the user space task and restart it where it left off on the next CPU core. This division of labor between the kernel-space dispatcher and the user-space scheduler is a pattern used commonly within Barrelfish, with more core functionality being handled in user space than would normally be found in a Linux system.

Like thread scheduling, address space migration is also handled in user space. A user space process called a *monitor* is responsible for accessing remote state and maintaining replicated data. A process's virtual address space is one example of the data maintained by the monitor. Barrelfish supports shared address spaces between threads within a process. Sharing is accomplished by replicating hardware page tables across OS nodes.

2.1.2 Factored Operating System (FOS)

Factored Operating System (FOS) is an operating system designed for multi-core systems with thousands of cores [23]. It factors operating system functionality into a number of services and runs instances of each of those services on dedicated CPU cores. Services can be replicated on multiple CPU cores, enabling the system to exploit the spatial layout of cores in silicon to route an application request to the nearest core that provides the needed service. FOS also divides operating system services from applications, running them on separate CPU cores. Each core runs a microkernel, which hosts either a service or application code. A service provided by a core can be consumed by software on any other core over a networked interconnect. Applications interact with the remote system services through a local proxy that tunnels requests through the microkernel, which in turn handles the inter-core communication that is required to service the request.

There are a number of advantages to organizing a computer system in this fashion. Because the application code and the OS code execute on separate cores, there is no potential for the two to interfere with one another. Additionally, because the number of cores providing a specific service can increase with increased core count, OS scalability is inherent.

Applications in FOS execute on one or more cores and are scheduled by a dedicated scheduling/placement service. Applications running across multiple cores share pages, and therefore a virtual address space. Page sharing is facilitated by the OS layer, but the OS layer itself does not utilize shared pages. It exclusively uses explicit microkernel messaging.

2.2 Cluster Operating Systems

2.2.1 Disco

Disco [7] is a scalable virtualization-based environment used to host client virtual machines on hardware the virtual machine was not initially built for. It seeks to solve the problem that

hardware vendors often encounter when introducing new architectures, namely the hesitance of system software vendors to exert the effort necessary to produce a port. Disco introduces a layer of software on top of multiprocessor hardware which virtualizes the hardware. On top of that layer, one or more virtual machine monitors host commodity operating systems. Those client operating systems interact in a networked fashion to form a cluster. Because Disco controls hardware access, it can transparently force exploitation of hardware features while removing the need for software modifications in the client system software to support those hardware features.

One example of the type of optimization work done by Disco is processor virtualization. Disco structures virtual processors to present common processor architecture interfaces to client operating systems. As CPU usage changes across client operating systems, Disco shuffles virtual processors between those clients in order to more efficiently match processing load to processing hardware.

Another type of optimization performed by Disco involves shuffling memory between virtual machines to satisfy a client's memory requirements when another virtual machine is not using all of its available memory. The monitor is also able to present a uniform memory access (UMA) like view of memory on NUMA machines by strategically allocating memory to virtual machine clients and dynamically migrating and replicating pages. This feature extends its portability support to client operating systems that expect to run on UMA machines.

2.2.2 Hive

Hive is an OS designed for shared memory multiprocessors and is implemented on a Stanford FLASH multiprocessor[8]. It is comprised of numerous interconnected kernels. Each kernel is referred to as a cell and contains a set of CPUs, a designated region of memory, and I/O devices. Hive aims to create a fault tolerant environment in which faults are restricted to single cells and do not affect the operation of other cells. It is also focused on providing a single system image similar to those provided by SMP systems. Cells cooperate in order to form this unified user environment.

Communication between cells is necessary to provide user functionality. Hive prohibits direct write operations by a cell into a target cell's OS internal memory region and uses hardware memory firewall capabilities to guard cells from writes by errant cells. Rather than using direct writes, Hive uses remote procedure call (RPC)s for most inter-cell communication. However, Hive does allow direct reads from other cell's memory regions. When doing direct reads from another cell's memory, the reading cell is responsible for sanity checking the data that it read, doing deadlock detection and avoidance, and otherwise protecting itself against inconsistent, broken data that may be present in the target cell.

While inter-cell write operations to OS internal memory is prohibited, Hive allows inter-cell

writes to user pages. This is necessary in order to exploit the performance advantages of shared memory. User pages that are used only by processes local to a cell are protected against writes. The remaining pages are dynamically write protected. Negotiation between cells result in decisions about whether to protection these areas. Hive also protects against faulty user page writes by other cells by assuming that any writable user page has been corrupted when a cell is detected to have faulted. Potentially corrupt pages are unloaded to protect the system from damage.

Hive supports multi-threading. A process with threads on multiple cells at the same time is referred to as a *spanning task*. Hive keeps processes address space coherent across a thread group, and supports thread migration for the purpose of load balancing. Interestingly, the authors note that there are complex trade-offs involving spanning tasks and fault containment. For example, tasks borrow pages from cells as the tasks migrate between those cells, thus increasing the number of cells on which the task depends. If a cell were to fail, all of the pages borrowed from that cell would no longer be trusted.

2.3 Linux Approaches to Multikernel Design

2.3.1 NoHype

NoHype is a cloud computing solution that focuses on virtual machine (VM) security [20]. Cloud service providers sell a VM execution environment. Multiple cloud service customers' VMs often coexist on servers maintained by the service provider. Research has demonstrated that it is possible for software running in a VM to break out of its virtual environment and into the host machine. Using these techniques, a malicious user can gain access to VMs owned by other customers of the cloud service, thereby compromising sensitive data and services. NoHype is focused on providing assurance that one VM cannot affect the availability of another VM, access data or software on another VM, or use side-effects of another VM's operation to deduce anything about what that VM is doing.

NoHype is a virtualization solution without an active virtualization software layer. NoHype accomplishes this by taking steps to partition server hardware resources between VMs and allowing VMs to run native on those resources. NoHype establishes a one VM per CPU core model. In a multi-core environment, this allows multiple customer VMs to co-exist on a given server. However, NoHype removes communication between VMs because the number of shared resources is reduced. Hardware-enforced memory partitioning sandboxes a given VM to its designated physical memory regions. Hardware memory separation is performed by the multi-core memory controller (MMC)'s extended page tables (EPT) feature, which adds another logical layer to the virtual to physical address resolution process. This removes the ability of a compromised VM to read from or write to another VMs memory or affect its operation. Additionally, hardware enabled virtual I/O devices supply I/O capabilities to

VMs, while segregating their use of those resources and assuring access to them based on time slotting.

VM management and operation are decoupled in NoHype. The NoHype system manager executes on a dedicated core and is responsible for starting and stopping VMs. Per CPU core management software runs on a core before and after the client VM, but never during. Additionally, many of the hardware separation features support fault delivery. When a VM attempts to access resources outside its own resource partition, a fault is delivered to the NoHype system manager, e.g. if a VM attempts to access a physical address that is outside the address range allocated to it, the system manager will be alerted. This allows the cloud service provider to detect attempted system exploitation.

2.3.2 coLinux

cooperative Linux (coLinux) is a virtualization solution that breaks the normal VM to host virtual machine monitor (VMM) power relationship in which the VM is unprivileged relative to the host [1]. Rather in a coLinux system the VM is as privileged as the host, running in ring 0, and context switches between VM and host are explicit. In this arrangement, the VM and host are cooperative. This approach trades increased performance and sustainability for decreased security and stability. The VM executes directly on the CPU, so no overhead penalty is paid for hardware virtualization. Sustainability increases come from the minimal nature of the modifications necessary to commodity operating systems to add support for cooperation, making modifications to the kernel proper and adding a kernel module. The security penalty comes from the fact that the VM has privileged access, and can therefore access arbitrary host resources. Stability of the host system can be adversely affected due to the cooperative nature of a system like this. If the VM were to fail to yield control back to the host due to a failure, the host system would starve of CPU access.

coLinux accomplishes RAM division between host and VM in an interesting way. The host allocates memory for use by the VM, and holds that memory never releasing it. That memory is used exclusively by the VM, and is never touched by the host. Because the host does not free it for the duration of the lifetime of the VM, the VM is guaranteed to always maintain control over that specific region of memory.

coLinux also handles context switches very elegantly. A context switch is comprised of instruction pointer modification and a page table swap. However, there is no instruction to do both tasks at the same time. The two operations must be split into two steps. This is problematic because the virtual address space in the two contexts is not likely to align, causing memory access faults. The solution employed by coLinux introduces an intermediate address space which maps the context switch code into two memory regions. It is mapped into the virtual address range it is expected to be in for both the host and the VM. When the context switch occurs, the starting context switches out its address space for the intermediate address space. It is then able to proceed to the next instruction, because the memory out

of which it is executing is still valid due to the fact that the context switch code is mapped into the intermediate memory map. The instruction pointer can then be changed so that the other context is actively controlling the last part of the context switch. The new context then swaps its address space in, replacing the intermediate address space and completing the context switch.

Context switching in coLinux also involves substituting the interrupt vector table. The interrupt vector table for the VM is constructed such that the interrupts that it needs to operate are routed to the VM, while the remaining are forwarded to the host through a proxy ISR. The proxy ISR conducts a context switch and invokes the host ISR.

2.4 Software Partitioning of Hardware

2.4.1 Twin Linux

Joshi et al. observe that there are three distinct types of computing - those tailored for user interaction, for servers, and for real-time applications [19]. Twin Linux seeks to create an environment in which all three computing can be done at once by running distinct kernels on each CPU core in a multi-core system simultaneously. Each kernel can be optimized for one of the three types of computing.

In Twin Linux, there is a single bootstrap processor (BSP) and all other processors are considered application processor (AP)s. The BSP is brought up first and hosts the GRUB bootloader while all of the APs are held in a halted state. All AP local advanced programmable interrupt controller (LAPIC)s are configured to ignore all inter processor interrupt (IPI) vectors except the INIT and STARTUP IPIs. In order to boot the APs, the BSP follows the Universal Start-up Algorithm as specified in the Intel MultiProcessor Specification [17]:

```
BSP sends AP an INIT IPI
BSP DELAYs10mSec
if APIC VERSION is not an 82489DX then
    BSP sends AP a STARTUP IPI
    BSP DELAYs 200uSec
    BSP sends AP a STARTUP IPI
    BSP DELAYs 200uSec
end if
BSP verifies synchronization with executing AP
```

Some work must be done in order to facilitate the co-existence of two distinct operating systems in a dual core system. First, RAM must be divided between the operating systems. In Twin Linux RAM is divided evenly between resident operating systems. Second, SMP

support must be disabled. Disabling SMP support ensures that the kernel does not attempt to wake up other CPU cores as it boots. Third, hardware sharing is set up - one kernel masks all PCI devices, while the other masks all PCI-X devices. This allows for the installation of two different network adapters, each of a different type - PCI and PCI-E - ensuring that each kernel will have access to one of the network adapters. Lastly, video RAM is divided between the two operating systems, allowing one to display on the upper half of the monitor, and the other to display on the bottom half of the monitor.

2.4.2 SHIMOS

Single Hardware with Independent Multiple Operating Systems (SHIMOS) is a solution for hosting multiple operating systems on a single multi-core hardware platform [22]. In SHIMOS, hardware resources that are typically shared in a strict SMP environment such as memory, and CPUs, are instead partitioned among operating system instances. Each resident OS instance is allocated a subset of the available CPUs, and its own portion of memory. Additionally, each resident OS gets its own network interface card (NIC), hard drive, and hard drive controller. This solution results in the ability to host multiple OSs directly on hardware without the aid of a VMM.

SHIMOS is implemented as a modified Linux kernel and a kernel module that is used to start additional kernels. When the BSP boots, the number of CPUs subsumed by that OS is limited by the *maxcpus* command line input. The CPUs awoken during its boot process behave as an SMP grouping. That OS can then proceed to boot other OSs by invoking the Kernel Loader Module which is responsible for bringing up new OSs. The CPU that is woken by the Kernel Loader Module then acts as a virtual BSP, which in turn proceeds to subsume a number of additional CPUs into its kernel limited by its own *maxcpus* argument. In that fashion, a number of OSs, each running with a custom number of CPUs, are booted.

Kernel mode changes were made to support resource division among resident OSs. Memory usage changes were made in the kernel proper to keep kernel instances from interfering with one another. Changes were necessary to support shared interrupts so that devices, such as timers, which needed to be shared between OSs could work for all OSs. No modifications to Linux were necessary to support multiple NICs and hard disk drive (HDD)s, as each OS can be configured to use specific peripheral component interconnect (PCI) devices.

2.5 Compute-Node Kernels

Compute-Node Kernels is part of a systems architecture in which separation of concerns plays a significant role [21]. Blue Gene/L is an IBM server comprised of a number of different types of nodes - compute nodes, I/O nodes, service nodes, front-end nodes, and file servers. Front-end nodes allow users to interact with the system. File servers house the files that are used

by the system. I/O nodes behave as gateways to file server and front-end services. Compute nodes are responsible for running application processes.

Each compute node is built around a Blue Gene/L ASIC which integrates two PPC440 processors with dual FPUs. Each PPC440 has its own L2 cache; each of which shares SRAM. They also share an L3 cache which interposes 4MB of eDRAM. Blue Gene/L is divided into partitions, which are 8x8x8 collections of compute nodes. Each partition is capable of running exactly one job at a time, each of which may have some number of processes within. Processes within a job are bound to a single compute node. Each compute node runs a compute node kernel (CNK). A CNK is a small operating system which is responsible for four things. For each job it creates an address space for the process that it is about to host. Address spaces are fully constructed at this time to give the new process access to its code and the shared data set - no paging is done once the application gains execution. It then loads executable and data contents into that address space. Finally, it transfers execution control to that process, switching from supervisor mode into user mode. Once this happens, its only remaining responsibility is to service system calls as they are made by the application.

Some system calls that Blue Gene/L support are very simple, such as getting the time. Those simple system calls are serviced directly by the CNK. Others are more complex and must be delegated to the I/O nodes. An I/O node, after receiving a system call to do something, proceeds to acquire data from a file server or interact with the front-end nodes. The results of that interaction are then returned to the CNK, and finally reported up to the application as necessary. While this happens, the process pends.

Processes in Blue Gene/L run in parallel and operate on shared data. Early implementations supported running the same code in each process against shared data sets. This allowed a user to process large amounts of data using parallel processing algorithms. Later iterations allowed concurrent processes to execute different code, broadening the systems processing capabilities.

This design has numerous advantages. The division of labor that is present in this architecture is conducive to simple component design, since each component is responsible for a small number of tasks. It is also extremely scalable, as the software design is so closely tied to the hardware design.

2.6 Operating Systems for Emerging Exascale Multi-core

2.6.1 Tessellation

Tessellation, developed by Colmenares et al., applies adaptive resource-centric computing (ARCC) at the system level to dynamically evolve the OS to application requirements [12]. In Tessellation, software components are compartmentalized into performance isolated and security protected cells. A cell contains either an application or a system service. Services provide quality of service (QOS) contracts to service clients. For example, a network service provides networking capabilities to an application or another service. A QOS contract exists between that network service and each of its clients, and the service works to honor each of contracts. Cells communicate through the use of a channel abstraction.

Tessellation implements gang-scheduling, a scheduling paradigm in which one or more separate software components that are working together are scheduled to execute simultaneously for execution on different CPU cores. Scheduling is done in two stages. The first stage of scheduling matches resources to cells. When a resource is given to a cell, it can be used exclusively by that cell. This eliminates resource competition. The second stage of scheduling happens within a cell to match specific work to a time slot.

The systems ability to adapt resource allocation to service resource needs is implemented by a system service called a resource allocation broker (RAB). A RAB matches resources to services with the goal of maximizing system wide metrics such as the number of QOS requirements met. While a service is responsible for meeting its QOS contract, its ability to do so is affected by the resources that it has been allocated by the RAB at a given time. When a cell starts, it communicates its performance metric goals to the RAB. The RAB then continually compares actual performance to those goals, as well as to global metrics to maximize system performance, and adjusts resources as needed.

2.6.2 Tornado

Tornado is a shared memory multi-core OS that seeks to reduce resource contention by increasing locality of system services. [16]. Tornado is an object-oriented OS and is designed to exploit the locality inherent to objects to reduce contention. The object model is designed to directly reflect what an application is interacting with, e.g. processes, files, caches, hardware address translator (memory management unit (MMU)). Each of these items have corresponding object representations within the system software. Locks are used within objects. So when an application uses a system call to interact with a file, for example, only that files internal locks will be used for serialization, as opposed to using more global locks as is common in other shared memory OSs. As a result, if two processes interact with two

different files, there will be no contention. This, however, does not solve the problem of locking in shared objects such as shared files. To address this, the author introduces the notion of a clustered object. A clustered object is one that may be accessed by multiple threads and CPUs. A clustered object can be represented in different ways. Some clustered objects are represented by a single representative object that is shared across all CPUs. In this case, the shared object behaves a lot like a normal shared memory object with respect to locking and use patterns. Other clustered objects are represented by multiple representative objects - one per CPU, or one for each grouping of CPUs. For these types of clustered objects, different strategies are employed to ensure consistency.

Because there are often multiple threads in a process, and threads may execute on different CPUs, the Process object is a clustered object. The Process object is replicated per CPU. However, one Process object is dominant, and operations against a Process object are directed to the dominant object for handling. The non-dominant Process objects behave as proxies to the dominant Process representative. Other types of clustered objects, like the list of virtual memory areas, are loaded on demand on a CPU. Those objects are migrated with threads when they are migrated to other CPUs to maintain the benefits of locality. This results in multiple replicated clustered virtual memory area objects in use when threads in the same thread group exist on different CPUs. By replicating the virtual memory objects, page faults can be handled locally without use of heavy cross-CPU locking. However, adding and removing virtual memory area contents becomes heavier in this case, as coordination between replicated objects must occur.

2.6.3 Corey

The authors of the operating system Corey[5] argue that performance bottlenecks may be caused by the operating system's lack of knowledge about what is best for a given application. They propose inverting the relationship between application and certain OS resources, allowing the application to control how the system software shares resources. They introduce three operating system abstractions in the Corey operating system. Each of these abstractions are controlled by applications that are hosted by Corey, since it is presumed that they know best how to optimize themselves.

The *address range* abstraction allows an application to control which parts of its address space are shared between tasks, and which are private to a given task. An address range represents a set of virtual to physical address mappings, and has data associated with it specifying how it is to be shared. The data stored in an address range is reflected in the hardware page tables.

The *kernel core* abstraction allows applications to control which cores are dedicated to which system functionality. For example, hardware drivers can be given affinity with a specific CPU core. This removes lock contention between CPU cores that are running driver code. The driver that runs on that single core can be interacted with via shared-memory IPC, but the

driver executes only on the selected core.

The *share* abstraction allows applications to share references to kernel objects, such as file descriptors. Each application has a set of shares, and it has the power to decide which other applications it shares each share with. This is an interesting abstraction, as it allows for dynamic scoping of kernel-hosted objects that are typically process-local.

Corey is a research operating system, and has not had the opportunity to mature to the point where it is reasonable to compare it with a commodity operating system such as Linux. However, many of the concepts that were introduced by the Corey team could be used within those commodity operating systems.

2.6.4 Sprite

Sprite is an operating system designed to identify idle machines on a network, and migrate processes to those machines[15]. When a process migration occurs, the process that was moved still has access to the same files, virtual memory, and devices that were present on its home machine. It also keeps its pid. The only observable difference is in machine load, where the source machine's load decreases, and the destination machine's load increases.

The authors of Sprite identify four trade-offs when implementing process migration: transparency, residual dependencies, performance, and complexity. *Transparency* is a measure of how differently a process must act after a migration. If a migration is very transparent, it acts no differently. *Residual dependencies* are dependencies that are created between a process and a specific host. If a host-specific resource must be exposed to remote processes, it has residual dependencies. *Performance* is how efficient the migration mechanism is. A very high performance migration mechanism creates no overhead during the migration, and the process appears to perform exactly the same as if it were never migrated. *Complexity* is a measure of how much of the operating system must be aware of migrations in order to operate. The authors note that these factors are in conflict, and therefore not all can be optimized at the same time. In Sprite, transparency and performance were optimized at the expense of residual dependencies and complexity.

When a migration occurs, the process control block (PCB) for the migrated process is partially replicated from the source to the destination, including register states. However, the PCB is not discarded on the source machine. Instead, it remains and stores state. The PCB for the migrated process on the destination machine is abbreviated, and is used mostly for look-ups and scheduling.

Sprite's solution to address space migration makes use of a network file system. When a process is migrated, it flushes its address space to a file on the network drive. When the process resumes execution at its destination, it loads pages from that network file on-demand. This system uses the file server's memory to cache address space information. In many cases this completely eliminates file operations, as the file server's memory can be used exclusively

except if cache space is exhausted. The drawback to this mechanism is that more network traffic is created than would be necessary if a direct transfer was made from the source machine to destination machine. Another optimization made by Sprite is to prefer *exec* time to do process migrations. When this happens, there is no address space to migrate. Because of the complexity associated with migrating shared pages and keeping them coherent across workstations, migrations that would create this situation are disallowed.

2.7 Scalable Virtual Memory Subsystems

2.7.1 Bonsai

Bonsai is a scalable virtual memory subsystem that seeks to reduce address space maintenance overhead within an operating system[10]. Most operating systems in wide use protect their virtual memory map data structures with heavy per-memory map locks that must be acquired before making modifications. They must also be acquired before reading the contents of the memory map. In Bonsai, the notion of read-copy-update (RCU) was applied to virtual memory indexing to reduce contention and allow modifications to occur concurrently with read operations. The data structure used to implement this indexing mechanism is called the Bonsai tree, which is an RCU-based tree structure containing memory mappings. Read operations can occur without lock. A write operation constructs a new tree, then atomically swaps out the pointer to the tree root, leaving the new tree in place of the old. A number of optimizations are implemented in Bonsai on top of this basic concept that in certain cases allow it to forgo the entire re-building of the tree, and instead allow for the use of pointer redirection. For example, a node that has the same children before and after an operation, can be reused. This eliminates the allocation of the new version of that node, as well as the deallocation of the old node.

This method of virtual memory indexing was implemented in a Linux environment, and evaluated against a number of benchmark workloads. Benchmark results showed between 70% and 240% performance increase in an 80 core machine over an unmodified Linux kernel.

2.7.2 RadixVM

RadixVM is a virtual memory subsystem built by Clements et al. at MIT, that achieves perfect scalability of memory manipulations for operations that involve non-overlapping memory regions[11]. RadixVM uses novel scalable reference counting data structures to track page usage for determining when pages are free for reuse. For indexing virtual to physical page mapping information, a fixed-depth radix tree is used. The use of a radix tree allows for the compression of large regions of unused virtual address space. Within the radix tree, there is a single object representing each used virtual page, which holds a pointer to the physical

page that is allocated to it. The reference counting data structures are used to track which nodes are and which are not in use so that the node can be removed from the radix tree when a page is no longer in use.

Because of the structure of the data, each node, which represents a virtual page, can be locked independently. During mmap operations, some of the leaf nodes may not exist yet. When this is the case, RadixVM acquires a lock on the farthest inner node that does exist, and propagates the lock to new leaf nodes as it unfolds the tree. Each time it pushes the lock out to a new leaf, it unlocks the parent lock in order to allow that node to now be reclaimed by another task. When the mmap operation is complete, it unlocks the only remaining lock that it holds.

During page faults, a lock is acquired on the corresponding leaf node, and a physical page is allocated to it. The lock is then released.

2.8 Limitations of Previous Work

2.8.1 Single system image

Some systems, such as Twin Linux, SHIMOS, and coLinux segregate kernels within multi-core environments, allowing them to execute simultaneously and independently. However, they do not make the next step towards a multikernel - stitching together a single system image to create a unified user space running across those independent kernels.

2.8.2 Address space consistency

Some operating systems, such as the current version of Barrelfish (change set 1298:12e657e0ed48) do not currently maintain a consistent address space across OS nodes for distributed thread groups. Rather, in Barrelfish, when a thread is migrated, its address space is migrated in bulk with it. However, threads might modify their address space subsequent to a migration and when they do the modifications are not made visible on remote OS nodes. This is not a problem for workloads that do not create, destroy, or modify mappings after they begin execution. This does not suffice, however, for general purpose computing environments in which a consistent address space across all threads of execution within a process is required for correctness.

Chapter 3

Linux Concepts

3.1 Linux Execution Contexts

Linux is a multiprogrammed environment in which multiple execution contexts coexist. Each context is scheduled for execution by the scheduler according to some scheduling policy, and allocated time slices during which it has the CPU. In Linux, every execution context is described by a structure called a *task_struct*. All information about an execution context can be found in that structure. When an execution context is restored for execution, it is restored from the state information that is accessible through its task_struct. When its time slice is over, its state is saved to that task_struct. A tasks state includes register contents, memory mapping information, file descriptor entries, locks owned, a signal handler list, task hierarchy information, state flags, etc.

An execution context can be logically categorized as either a *process* or a *thread*. A process is an instance of a program that is currently executing [4]. A thread is an execution context that is a logical component of a process, where all threads within a process share a common address space and other common resources. User-space libraries (pthread) exist to help programmers create multi-threaded programs that make use of multi-threading capabilities that are supported by the kernel. Within the Linux kernel, each process or thread is tracked as an independent execution context that has its own corresponding task_struct. User-space threading libraries make use of a feature in the Linux kernel that allows multiple execution contexts to share resources. When a new thread is created, the threading library specifies to the kernel that the new execution context must share its memory map, signals, and open file descriptor list with its parent process. Execution contexts that are created in this way are said to be in the same *thread group*.

A task can exist in one of five distinct states: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, TASK_TRACED. A task in the TASK_RUNNING state is one that is eligible for execution. It is either executing, or waiting to be scheduled for

execution. A task in the TASK_INTERRUPTIBLE state is not eligible for execution. It can be placed back in the TASK_RUNNING when certain interrupt or signaling conditions are met. A task in the TASK_UNINTERRUPTIBLE state is subject to the same execution rules as one in the TASK_INTERRUPTIBLE state except that the kernel will not place it in a runnable state in response to any interrupts or signals. A task in the TASK STOPPED state has been stopped. A task in the TASK_TRACED state has been stopped by a debugger.

When a tasks time slice is over or the task voluntarily yields its time slice, the kernel preempts its execution. Its state must be saved to its task_struct so that the task can later be resumed where it left off. There are always two current stacks, a user-mode stack for use while in user mode and a kernel-mode stack for use while in kernel mode. When the user-mode code is preempted and the kernel takes execution, the register values of the previously executing user code are placed on the kernel stack. A reference to that saved register state can be acquired through the task_struct by passing the task_struct into the macro *task_pt_regs*. When it is time to restore that user mode task, those register values are restored from the kernel stack.

Each task state consists of not only a user-space state, but also a kernel-space state. During a context switch the kernel mode state is swapped out as well as the user mode state. During the call to *schedule()*, two major changes are made: the memory map of the next task is activated, and the kernel stack and hardware context are swapped out for that of the new task. When the next kernel stack is swapped in, the user mode context is also swapped in by virtue of the fact that its state is stored on the kernel mode stack. When the kernel subsequently restores user mode execution, the appropriate register values are therefore used.

3.2 Task Creation

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. When a process decides to create a new task, it undergoes a process called *forking*. A fork operation consists of creating a duplicate of the calling task, and ensuring that it is schedulable. The calling task is then the parent of its newly created copy, and the copy is the child of the calling task. Once the new task is created, it can execute independently of the parent task. In many cases, in user-space, the new task invokes the *exec* syscall when it resumes execution to change into a completely different program. The kernel then selects a binary format handler to pull the contents of the specified executable file from the file system, and construct the new programs virtual address space according to the contents of that executable file.

Forking is initiated by a user-space task by calling the *fork* syscall. Once in kernel space, a function called *do_fork* is invoked, which implements the fork operation. A kernel-mode task instead calls the *kernel_thread* function to create a new kernel task. One of the parameters of the *do_fork* function is the register set of the process that is to be forked. That state

information is used when creating the duplicate task. The newly created task will be resumed from the exact same place in the program as the parent task left off, which is captured in the register values. In order to provide a starting point for the newly created kernel task, *kernel_thread* creates a temporary register set, and initializes the member variables. The function that makes up the kernel task's body is passed into the *kernel_thread* function. The address of that function is assigned to the *si* register of the newly created register set. The instruction pointer in the newly created register set is set to the address of function *kernel_thread_helper*. When the task is created, it will begin execution at the beginning of the *kernel_thread_helper* function. That function, in turn calls the function at the address stored in its *si* register. Once all the register values are set, *kernel_thread* directly invokes *do_fork*, passing in the newly created register contents. When *do_fork* executes, it creates a task with the register contents as they were provided.

do_fork performs the work necessary to create the new task and add it to the scheduler's run queue. The first thing it does is it allocates a new task_struct. That task struct will eventually be placed in the task list, and scheduled for execution. But before that happens, it must be properly populated. *do_fork* takes as a parameter a set of flags that define which parts of the parent task should be shared with the child task. Items that are not shared, are copied to the child so that the child starts as an exact copy of the parent at the time of the fork operation. As fork constructs the child task, it consults those flags and switches its behavior based on which flags are set. The items that are optionally copied from the parent task include locks held, file descriptors, signal handlers, its memory map, the name-spaces it belongs to, and its I/O. Each of these items is represented by a data structure that is either copied to the child, or assigned to the child by reference. Items that are shared between the parent and the child are assigned by reference. In that case, a reference count specific to the shared data structure is incremented by one to indicate the number of users that data structure serves. Reference counting is done so that shared data structures are not deallocated when one of its users exits, only when all of them have exited. Once the child task has been constructed, it is placed on the runqueue and activated. The task is now eligible for scheduling.

3.3 Task Exiting

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 those events occur, the kernel responds by invoking function *do_exit* from the exiting task's context. This function is responsible for terminating the task and removing it from the system.

In order to accomplish this, it first sets flags indicating that the task is exiting. It then proceeds to systematically release the tasks hold on any resources it may have claim to, including its signal handler data struct, its memory map, any semaphores it holds, and file descriptors it is maintaining. Once this procedure is complete, it sets a flag indicating that

the task is dead, and invokes the scheduler. The scheduler will never again schedule the dead task, and the task_struct itself will be removed from the task list and garbage collected during the task switch process.

3.4 Scheduling

Linux is a time-sharing system. The Linux Scheduler divides time into timeslices per some scheduling policy, and gives execution to a task that it selects for the duration of each timeslice[4].

The scheduler is invoked when the function *schedule* is called from in kernel mode. This function selects the next task to execute, then does a task switch to activate it. When the task that invoked the *schedule* function is given the CPU again, it resumes execution after the call to *schedule*. Often a task invokes the scheduler when it is blocked, or needs to wait for some reason.

An alternate way of invoking the scheduler is by setting the *need_resched* flag in the current task's task_struct. That flag is always checked before returning to user space, and the scheduler is invoked if it is set.

3.5 Linux Memory Management

Paging is a system used in multiprogramming environments that allows each process to reference a complete address space without conflicting with the memory allocated to other processes [4]. A process never references physical memory directly by its address. Instead, a logical address is assigned to the physical address called a virtual address. Each process has its own virtual address space that covers the entire address range (without the kernel address range). A process always references memory by that memory's virtual address. A CPU hardware component called a Memory Management Unit (MMU) then transparently does a virtual to physical address look up. If a physical address is found to correspond to the accessed virtual address, then the operation that was going to be done on that address (whether that operation is a read, write, or execution), is done on the correct corresponding physical address. If no physical address is mapped to the virtual address being addressed, then a fault occurs.

The mappings between physical and virtual addresses are stored in a series of hierarchical page tables. There is one set of page tables for every unique thread group that has its own memory map. Only one set of page tables can be active at a time. The page tables that are currently active are the page tables corresponding to the currently active process. A page table is activated by writing the physical address of its PGD into register cr3. When a task

switch occurs, the newly selected task's page tables are installed in the MMU, and therefore activated in hardware.

Each page table entry, 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. Flags are set for read, write, and execute permission, as well as the type (if any) of caching to be performed. If a program attempts to use memory in ways that are not allowed by the permissions specified in the page table entry for a given address, a fault occurs. This is enforced by the MMU hardware.

In Linux, each task_struct contains a reference to a *mm_struct*. A mm_struct contains all of the information about an address space. This includes information about where the stack, heap, environment information, process arguments, and code live in the virtual address space. It also contains a reference to the PGD. When the kernel does a context switch, it swaps out the currently active PGD for the one stored in the next task's mm_struct, thereby changing the active virtual address space.

When Linux constructs an address space for a new process, it does not create all of the entries in its page table that the process may need. Instead, Linux creates a record of the fact that it owes memory to that process. Each of these records is held in a structure called a *vm_area_struct* or a virtual memory area (VMA). Every valid region of memory will have a corresponding VMA. Each VMA describes one contiguous region of virtual memory, as well as information about that region such as whether it is file backed (contents loaded from file) or anonymous (contents are not loaded from a file), permissions associated with that region, etc. No two VMAs for a given thread group describe overlapping regions of the virtual address space. A linked list of VMA structures can be found through the mm_struct for a given task. When a new region of memory is memory mapped into a processes address space, a new VMA is created for that region, and placed in that linked list. However, the page tables are not updated at that time. Instead, the kernel waits for the process to access that virtual area. When that happens, a page fault occurs. In response to a page fault, the kernel looks through all of the VMA's that exist for the faulting process. If one is found that encompasses the faulting address, the kernel then assigns physical memory to that virtual page, and adds an entry into the page table for it. If the mapping is file-backed, it loads the contents of the file into the newly assigned physical page. It then restores execution to user space where it left off, allowing it to try to access that address again. If a VMA is not found for the faulting address, a segmentation fault has occurred, and the process is killed.

When a process forks a child process, a new copy of the parent address space is created for the child. This is an expensive operation, as the contents of writable pages must be copied to new physical pages for the child process so that writes to the child's copy do not affect the contents of the parent's copy. To optimize this process, the concept of copy-on-write (COW) pages were introduced. When forking a child process, Linux marks writable pages as read only in both the parent and child page tables, flags those pages as COW, so that now the parent and child processes both reference the same read only memory in their page

tables. The first of those processes to attempt to write to a COW page will cause a fault and must *break* the COW page. Breaking a COW page involves copying the contents of the COW page to a new page and mapping that new page to the corresponding virtual page in the page tables. The page is then marked as read/write in both the parent's and the child's memory maps.

3.6 Linux User Mode Helper API

The Linux kernel is able to set up, execute, and interact with user-mode software through the use of its User Mode Helper API. The user-space processes that are created by this API exist outside of the family tree of init, and are instead children of kthreadd. The kernel uses this capability to accomplish many common tasks, including system shutdown, kernel module loading, and device hot-plugging [18]. In each of these examples, the kernel delegates work to user-space programs.

Chapter 4

Popcorn Concepts

4.1 Popcorn System Architecture

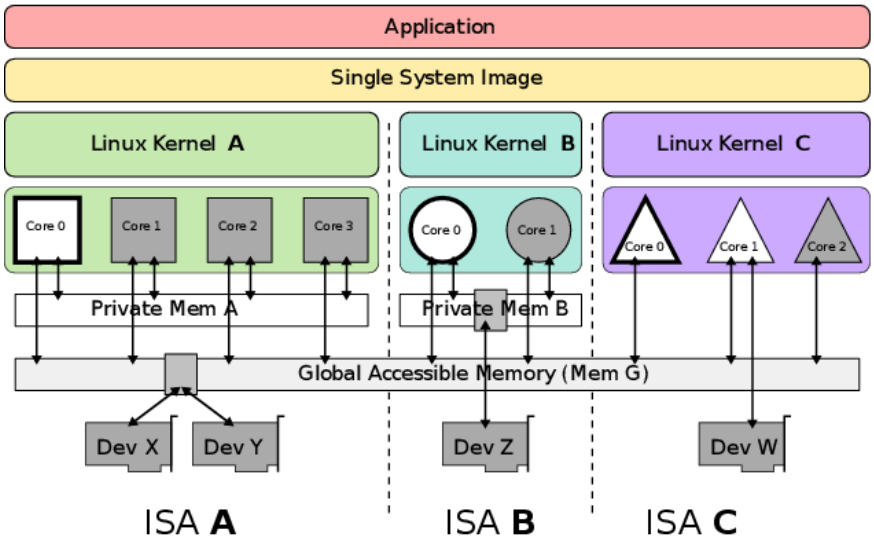

Figure 4.1: Popcorn System Overview

4.2 Hardware Partitioning

Hardware is partitioned into independently running Popcorn kernel instances. Computing resources can be assigned to a kernel instances in a number of ways.

Division by CPU Core In a homogeneous environment, it may be desirable to divide the system into kernel instances on a per-CPU basis. There can exist, at a maximum, one kernel instance per CPU in Popcorn. Clustering can be done to combine arbitrary CPUs into groupings, each of which shares a single kernel instance in SMP fashion.

Division by ISA In a heterogeneous environment, kernel instances can be assigned one to each ISA. This partitioning scheme places all cores that share a common ISA within the same Popcorn kernel instance cluster, each core acting as an SMP node within its kernel instance. This is a natural way to divide a heterogeneous system, allowing each kernel instance to have its own kernel compiled specifically for the ISA on which it runs. Further partitioning can also be done in this division scheme to divide the population of CPUs of a similar ISA into subgroups of arbitrary size.

4.3 Global Accessible Memory

Each kernel instance is assigned its own region of memory at boot time. This is the memory that it will use as it executes. It will not attempt to use another kernel instances allotted memory region. This is necessary in order to keep kernel instances from interfering with one another. There are exceptions to this division, however. When tasks are migrated between kernel instances, pages are often migrated as well. This is done in such a way that it is safe, as shall be explained in Chapter 5, and is strictly a user-space phenomenon. Another situation in which memory is shared is for creating a shared notion of time. This is also explained in Chapter 5. Lastly, a shared memory region is used for message passing between kernels.

4.4 Message Passing

Kernel instances, regardless of which subset of cores and ISAs they are running on, do not share internal data structures. Rather all communication is explicit via message passing. This is an absolutely critical component of Popcorn, as all communication and synchronization occurs through it. The task migration and address space migration and consistency algorithms developed for this thesis heavily leverage Popcorn's message passing capability.

4.5 Device Drivers

Device drivers are given affinity with a single kernel instance by design. The kernel instance that owns the device driver performs all driver operations. In the future, when a distributed

file-system is created for Popcorn, code on any kernel instance will be able to open a device node and manipulate it, but the work will be marshaled to the kernel instance that owns the driver for processing.

4.6 Single System Image

On top of kernel space, lives a single system image user-space. At the time of this writing, this single system image is not yet fully constructed. Its current state shall be described below. Once Popcorn is complete, all of the separate kernel instances shall work together to stitch this user-space together via message passing to present an interface that behaves exactly like the Linux user-space. Applications will not be able to tell the difference between an SMP Linux and Popcorn Linux system. Because of this, Popcorn will be able to host the wealth of Linux applications.

4.7 Applications Spanning Kernel Instances

Due to the single system image that is constructed by the cooperation of kernel instances, all applications that execute within user-space run seamlessly, regardless of which kernel instance is hosting the actual execution or even whether or not its tasks remain on a single kernel instance. Currently, only homogeneous x86 64bit environments are supported, but other architectures as well as homogeneous ones are being developed against.

4.8 Basic Building Blocks

In this section, the components that were built prior to the task migration mechanism that is the focus of this thesis, and upon which it is built, are described.

Individual kernel instances booted Hardware partitioning is supported. A kernel instance can be assigned to a single x86 64bit core, or to a cluster of x86 64bit cores. When the Popcorn environment boots, only the primary single kernel instance is booted. Once that kernel instance is booted, the secondary kernel instances can be booted at the user's discretion. Secondary kernel instances can be booted in any order, and it is not necessary to boot all kernel instances that the hardware supports.

Messaging Layer The messaging layer is a fundamental component of the Popcorn kernel. It is the only mechanism for interaction between kernel instances, except for a few exceptions.

The messaging layer, designed and implemented by Ben Shelton, and supported by the Popcorn team, exposes an API to the rest of the kernel components. This API allows Popcorn kernel instances to send messages to other Popcorn kernel instances. Each message has a type. In the context of this thesis, there are message types for mapping requests, mapping responses, task migration requests, etc. A Popcorn kernel instance can register associations between message types and a single function to handle incoming messages of that type.

The messaging API supports short messages, and long messages. Long messages are broken into chunks, where each chunk is the size of a short message. The larger a long message is, the more chunks are required to send that message. On the receiving side, a short message can be delivered to the handling function immediately. Long messages are reconstructed prior to delivery to the handling function, as they are must be delivered in their original form.

The sending and receiving kernel instances share memory in order to accomplish message transfer and signaling.

Message sending is accomplished by first atomically claiming a ticket, which signifies access to the transfer slot. A ticket is active when the assigned transfer slot is marked free for use. Once a ticket is active, the holder of that ticket places its message in the slot, marks it indicating that it is full, then generates an IPI to signal to the receiver that a message is waiting for it.

Message receiving is broken into two phases. A lightweight handler initially receives the message when the CPUs IPI handler is entered. This code executes in a bottom-half softirq context and is very minimal. This bottom-half code notifies the top-half that messages are ready for processing. The top-half, which executes within a workqueue context, is then invoked. In the top-half, the handler function that is registered for the specific message type is called, and the message is passed into that function as a parameter. Once that function has been invoked, and the message is freed, the message slot can be marked as empty and subsequently reused.

Chapter 5

Task Migration

5.1 Architectural Overview

A task migration involves the coordination of two kernel instances to transfer a task and its state from one kernel instance to the other. When that happens, the task that is migrated ceases to execute on the originating kernel instance, and resumes execution on the receiving kernel instance. The actions taken by the originating kernel instance are shown graphically in 5.1, and the actions taken by the receiving kernel instance are shown in 5.2. Tasks can be migrated any number of times between kernel instances.

In migrating a task, the originating kernel instance first transfers all of the tasks state information to the receiving kernel instance. The migrated task still exists on the originating kernel instance, however. That task is now placed in a sleep state and is considered a *shadow task*. It remains on the originating kernel instance as a shadow task until its *return disposition* is set. A task's return disposition specifies an action to be taken against a shadow task prior to waking it back up. A shadow task's return disposition is set in response to an event that has occurred remotely to the task on the kernel instance that it is currently executing on. That event can either be an exit event, or a return migration event. When an exit occurs, the return disposition is set to *exit* on shadow tasks associated with the exiting task on all kernel instances. When a migration is initiated that would cause it to execute on a kernel instance on which it has already executed, the return disposition is set to *migrate*. Once the return disposition is set for a shadow task, the shadow task is woken back up. When the scheduler schedules it for execution again it starts its execution by handling the return disposition. For *exit* dispositions, the task calls do_exit, killing the task. For *migrate* dispositions, the new task state is installed into the shadow task data structures, and it resumes execution in its new state when it is returned to user space. This is how a task is migrated back to a kernel instance on which it has previously executed. This migration process and the process used to migrate a task to a kernel instance on which it has not yet executed are covered in

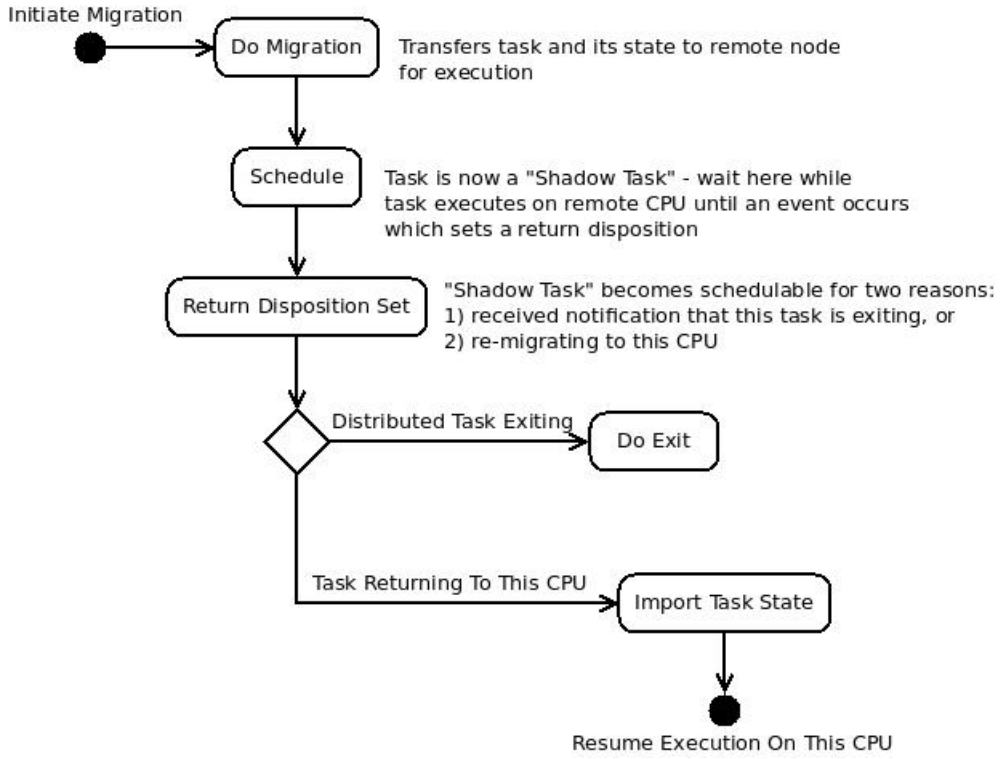

Figure 5.1: Migration - originating kernel instance

detail later.

5.2 Shadow Tasks

When a task migrates away from a kernel instance, the data structures that represent it on the originating kernel instance are not destroyed. Rather, they are left behind, and the task on that kernel instance remains dormant and is considered a shadow task. Shadow tasks serve multiple purposes.

As processes and threads execute they acquire resources such as file descriptors and locks. They may also spawn child processes or create sibling threads. When a task exits, work must be done to release resources and re-parent child processes. By turning a migrated task into a shadow task on the originating kernel instance instead of destroying it once the migration is complete, the release of its local data structures and resources is deferred. The shadow task then becomes the custodian of those resources, maintaining them on behalf of the migrated task. Those resources can then be migrated or otherwise used at some later point.

Additionally, the use of shadow tasks removes the overhead of destroying kernel data struc-

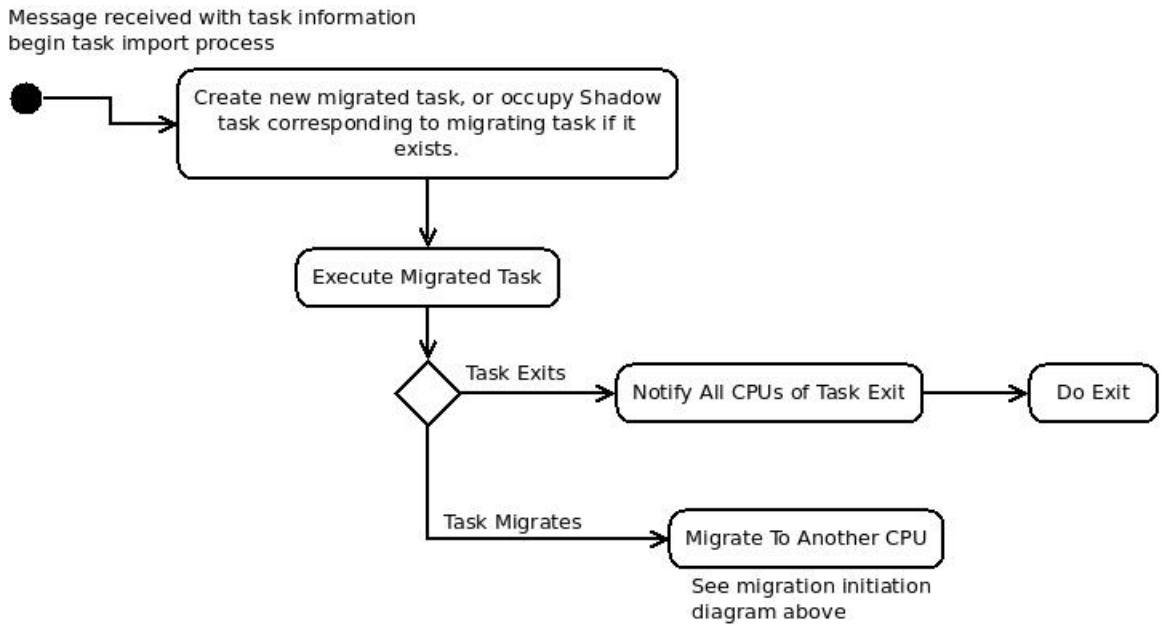

Figure 5.2: Migration - receiving kernel instance

tures when a task is migrated away from a kernel instance, and then recreating those same kernel data structures when that task is migrated back. When a task migrates back to a kernel instance on which it has previously executed, its current execution state is installed into the shadow task's data structures, and then that shadow task is awoken.

5.3 Distributed Task and Thread Group Identification

In Popcorn Linux, the task_struct data structure has been modified to track task identity across kernel instance by adding four fields. Those fields include the tasks kernel instance of origin, its PID on its kernel instance of origin, the kernel instance of origin of its thread group, and its task group identifier (TGID) on its kernel instance of origin. When a task is initially created, that information is stored in its task_struct. When that task is then migrated, this identification information is migrated with it. This information is used for addressing, and is included in all communications between kernel instances, as it is guaranteed to be unique for each task across kernel instances.

5.4 Forking Children

Propagating distributed thread group membership to children that are spawned to distributed parents is a simple matter of copying the parents distributed thread group identi-

fication, detailed above. This is done at *fork* time, when the child task is initially created. Distributed thread group identification information is only copied to the child if the child will be in the same thread group as the parent. Otherwise, new distributed thread group identification information is generated for the child based on which kernel instance the child was created on.

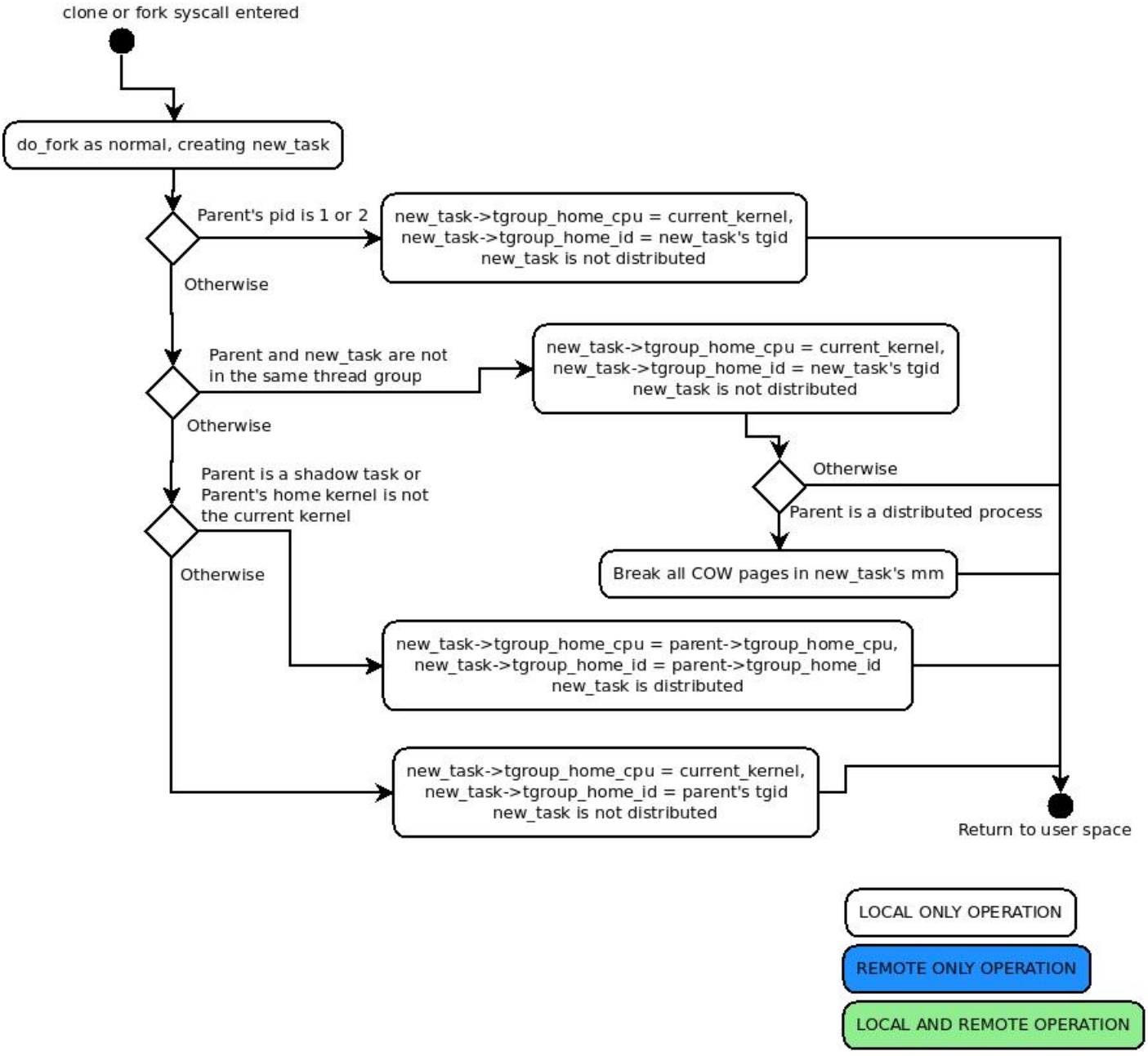

Figure 5.3: Task duplication hook

5.5 Task Exit Procedure

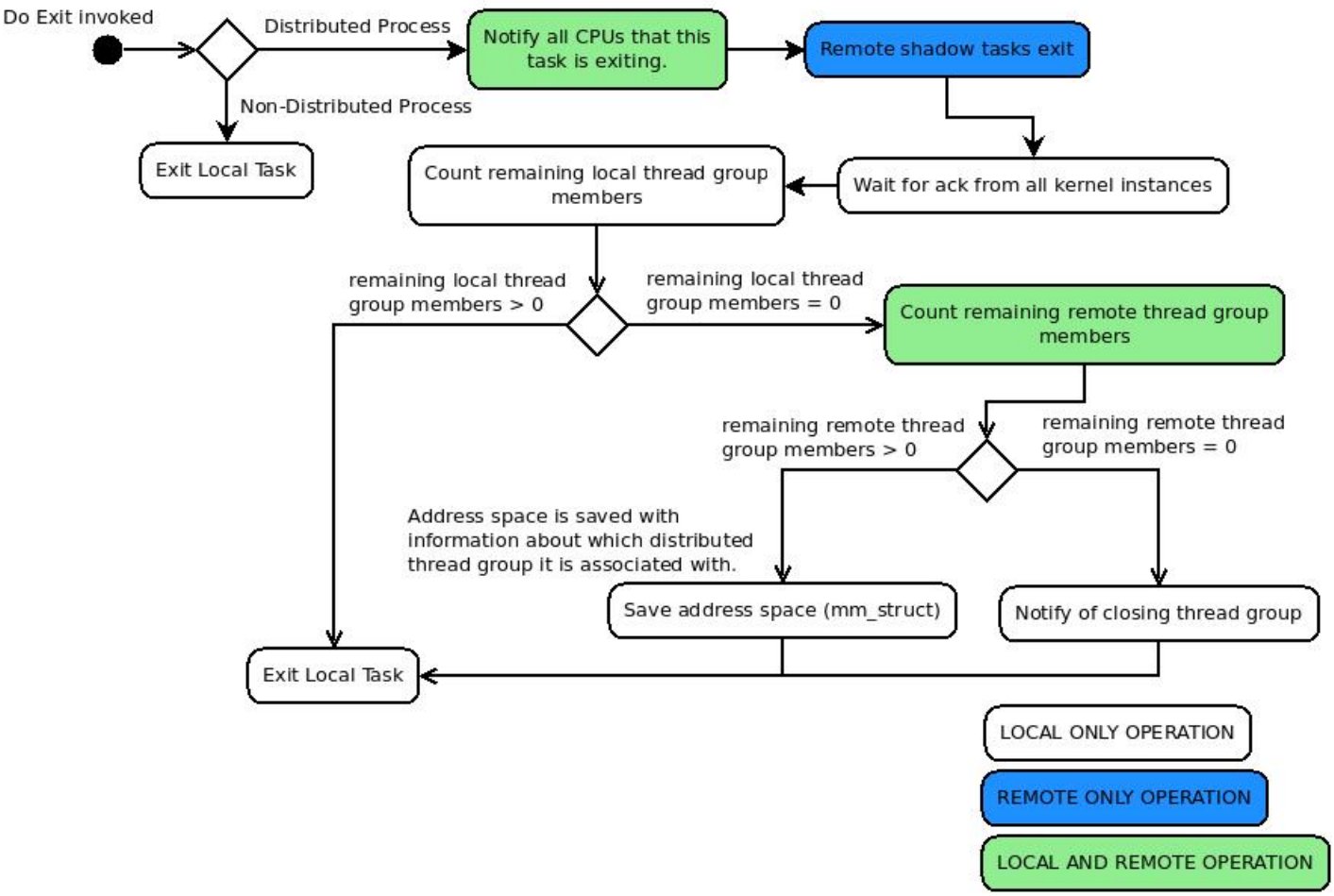

Figure 5.4: Task exit procedure

When a distributed task exits, not only does the currently executing task exit, but all shadow tasks associated with that task must also exit. The exit handler in the Popcorn Linux kernel has been modified to send a message to all kernel instances indicating which distributed task is exiting. Each recipient of that message then looks for any shadow tasks associated with the exiting task. For each shadow task that is found, a flag is set indicating that the return disposition of that shadow task is to *exit*, and the task is then woken back up. When the task resumes execution, it will be in kernel mode still. While there, it checks its return disposition. During this check it will find that it was resumed due to an exit event, it calls do_exit. This exits the task.

In addition to exiting the task, the kernel instance initiating the exit process described above checks to see if the exiting task is the last member of its local thread group. If so, it also

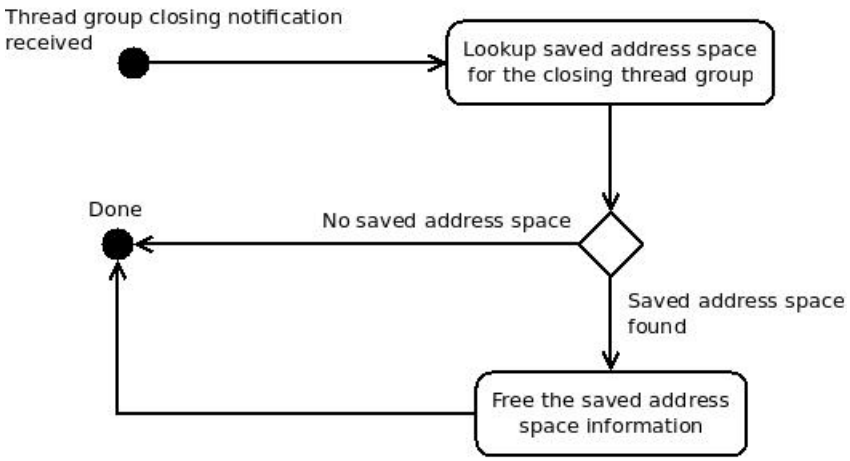

Figure 5.5: Response to thread group exiting notification

checks to see if it is the last distributed thread group member. If it is the last local thread group member, but not the last distributed thread group member, it saves its mm_struct. This is necessary to save any changes that were made to the address space locally that were never replicated remotely. This way, remote thread group members can still resolve new mappings that were created in the exiting tasks address space. If it is not the last local thread group member, this process need not be done since its address space will not be destroyed - as all members of a thread group share all address space data structures. If the exiting task is both the last local thread group member and the last distributed thread group member, the mm_struct is not saved. In this case it also sends out a notification to all remote kernel instances indicating that the thread group is closing. The remote kernel instances then use this opportunity to remove all saved address space information that they may have saved, as that information no longer needed.

5.6 Task Migration Mechanism

All task migrations are initiated by the originating kernel instance in response to either a scheduling decision or a syscall invoked from user space. When the migration is initiated, the steps described above are followed to change the current task into a shadow task. The task state is then communicated to the receiving kernel instance. The following information is sent to the receiving kernel instance.

1. Memory layout information
   (a) User space stack start and end addresses
   (b) Heap start and end addresses

(c) Environment start and end addresses
(d) Arguments start and end addresses
(e) Data start and end addresses

2. Task information

(a) General purpose and floating point register values
(b) PID of shadow task
(c) Kernel instance of Origin
(d) PID on kernel instance of Origin
(e) Kernel instance of Origin of Thread Group
(f) TGID on kernel instance of Origin
(g) Priority and Real time Priority
(h) Scheduling policy

The task migration procedure performed on the receiving kernel instance is slightly different when the received task has previously executed on that kernel instance. Much of the work that must done to import a task's state is already done in that case. As tasks migrate, a record is migrated and maintained that keeps track of which kernel instances this task has ever executed on. This record is consulted to select the correct migration procedure.

5.6.1 Migrating A Task To A New Kernel Instance

Kernel instances on which a task has never executed do not have an existing shadow task to host the received task. A new task must be created to host the migrated task's state. In order to create the new task, a new kernel thread is first created. Within that new kernel thread, a user-space stack is forged. Now that the new task has been created and capable of moving into user-space, the migrated task's state is transplanted into it.

All of the task information that was communicated to the receiving kernel instance from the originating kernel instance is now installed. This updates the task's task_struct, as well as the kernel stack, which contains the user space registers.

If the receiving kernel instance hosts other tasks in the migrating task's distributed thread group, the newly migrated task must be placed in the same local thread group as those other tasks. To accomplish this, first one of those other tasks is found. That task's grouping information is copied to the new task including its group leader, TGID, and parent. The new task is also removed from any sibling lists it may be included in.

Additionally, thread group members share many resources including their signals, signal handlers, memory map, file descriptors, and file system interfaces. The newly migrated task

must be set up to share those resources. Each of those resources is represented by a data structure, and the task_struct has a pointer to each of them. Those resource pointers are copied to the newly migrated task, and reference counts are increased as appropriate.

If there are no existing thread group members on the receiving kernel instance, one of two possibilities is true. Either there has never been a member of this distributed thread group on this kernel instance, or there has. In the case that there has previously been a member of this thread group on the receiving kernel instance, but all members of that thread group have already exited, that thread group's mm_struct will have been saved. That mm_struct will be installed as the newly migrated task's memory map. Otherwise, all of the memory layout information that was communicated by the originating kernel instance must be installed in the newly migrated task's mm_struct.

5.6.2 Migrating A Task To A Previous Kernel Instance

When a task is migrated back to a kernel instance on which it has previously executed, the hosting task 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, file system interfaces, and open file descriptors with the rest of its thread group. These objects do not need to be altered to in this case. All that needs to happen in this case is the task's current state information, which was sent from the originating kernel instance, needs to be installed into the shadow task, setting the appropriate user space register values.

Once the state has been installed, the return disposition is set to *migrate*, and the shadow task is awoken. Once awoken, the task returns to user space and resumes execution where it left off on the last kernel instance.

5.7 Address Space Migration and Consistency Maintenance

Unlike register values and state flags, a tasks address space is contained in a series of large data structures. It is not possible to migrate an address space by simply migrating its pointer to the PGD, since that would omit many important pieces of information contained in the virtual memory area (VMA) from the migration, force memory sharing between kernel instances, reintroduce shared locks, and remove much of the benefit of the multikernel. Instead, the contents of the address space must be partially or fully replicated across kernel instances.

Each kernel instance owns a portion of the physical memory present in the system. A task that originates on a given kernel instance is initially given memory from that kernel instance to execute out of and use. When a task migrates, all of its data and code live

in pages that are associated with a kernel instance on which it is no longer executing. In order to allow it to continue using those pages, two address space migration mechanisms have been implemented: up-front address space migration, and on-demand address space migration. Both mechanisms operate on the idea that pages from other kernel instances can be borrowed safely as long as they remain reserved on the kernel instance that owns them for exclusive use by the migrated task and its fellow thread group members. When a task migrates away from a kernel instance, a shadow task remains behind. That shadow task is the anchor that reserves the memory that that kernel instance has contributed to the migrated task and its distributed thread group. As that task continues to migrate to other kernel instances, shadow tasks are created on each visited kernel instance to reserve pages that were mapped while executing on that kernel instance. Mappings are migrated by communicating all pertinent information about the target mapping to the destination kernel instance, where it the mapping is then replicated - creating identical VMA entries and page table entry (PTE)s in the target task's memory map. When the task exits, all of its shadow tasks also exit. When that happens, all of the pages that were borrowed for the task and its thread group, and were held reserved by the shadow tasks, are released to their respective kernel instances for deallocation. This section provides a detailed description of how this mechanism works within Popcorn.

5.7.1 Address Space Consistency

Throughout the lifetime of a process, its address space can grow, shrink, and change based on its memory requirements. It was important to address this in Popcorn in order to create an environment that is consistent with Linux. As this address space change happens, Popcorn must take action to ensure that local changes are reflected in some way 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. They do not need to 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. For example, Figure 5.6 shows an example of how VMAs might be distributed across kernel instances. This example depicts two distributed VMAs, shown in light color. There are three kernel instances, each with a VMA that covers a slightly different area of the address space. This is a consistent state because if part of the local VMA is missing, but needed, it can be acquired from another kernel instance. Additionally, the type of all of the VMAs present on each kernel instance is consistent. There are, however, a number of inconsistent state that must be avoided. In the following we discuss the different types of inconsistency that must be avoided.

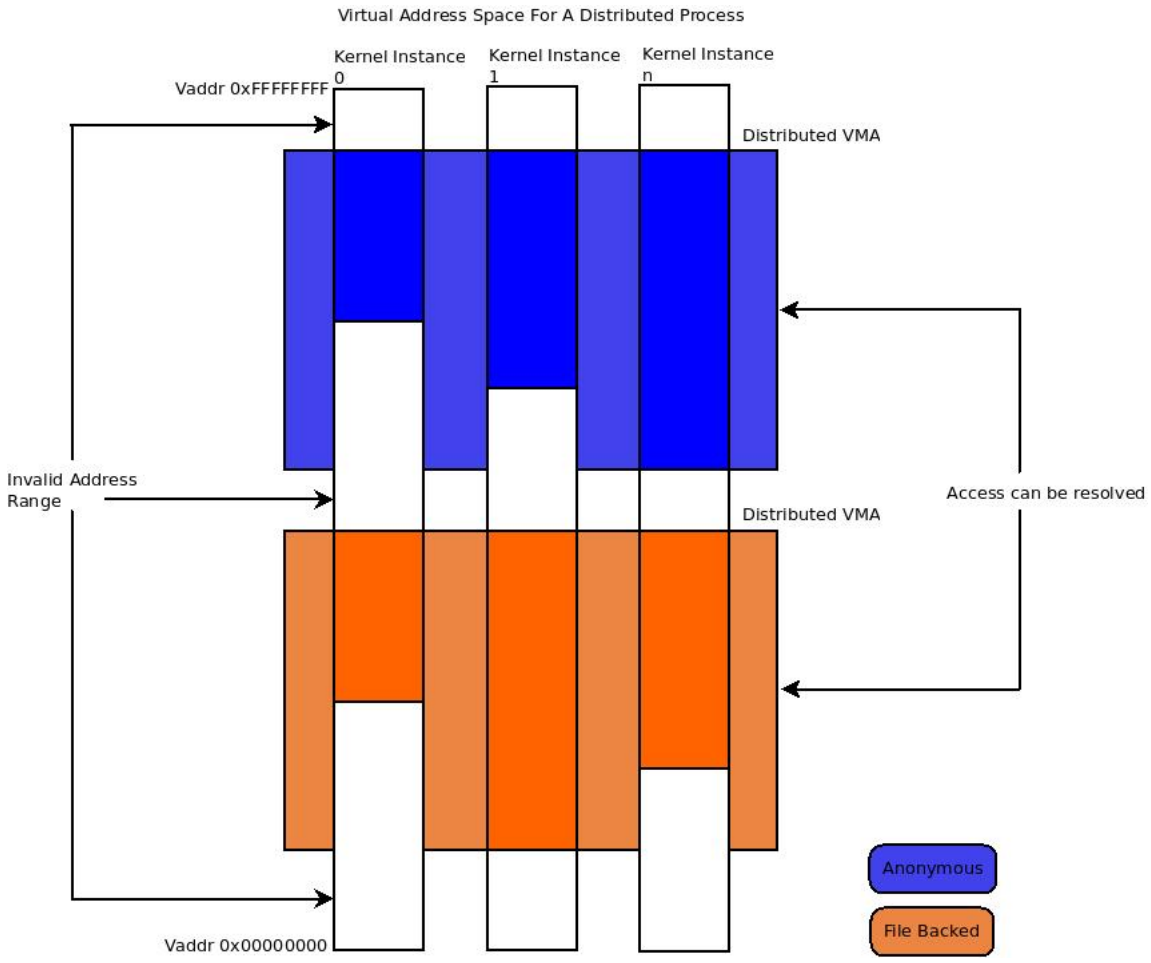

Figure 5.6: Two consistent distributed VMAs in a distributed address space. In light color the full VMA address range, in solid color the VMA mappings known by each kernel instance.

One type of inconsistency that must be avoided is one in which memory maps for the same distributed thread group on different kernel instances contain different physical address mappings for the same virtual page. In this case, tasks using those local memory maps refer to different physical pages of memory when addressing the same virtual address. This situation will break the applications ability to yield correct results, and sometimes cause segmentation faults or other runtime failures.

Protection mode inconsistency is a subtle but potentially dangerous problem. A page that is erroneously marked as write only, when it is writable on other kernel instances could cause segmentation faults that break the workload. This problem could be very difficult to find and debug and must never happen.

Another type of inconsistency is present when two corresponding mappings for the same distributed thread group on different kernel instances have different backing. A VMA can

be file backed or it can be anonymous. Two mappings with different backing types are inconsistent with one another. Two mappings backed by two different files are inconsistent with one another. Additionally, two mappings backed by the same file with different file offsets are inconsistent with one another.

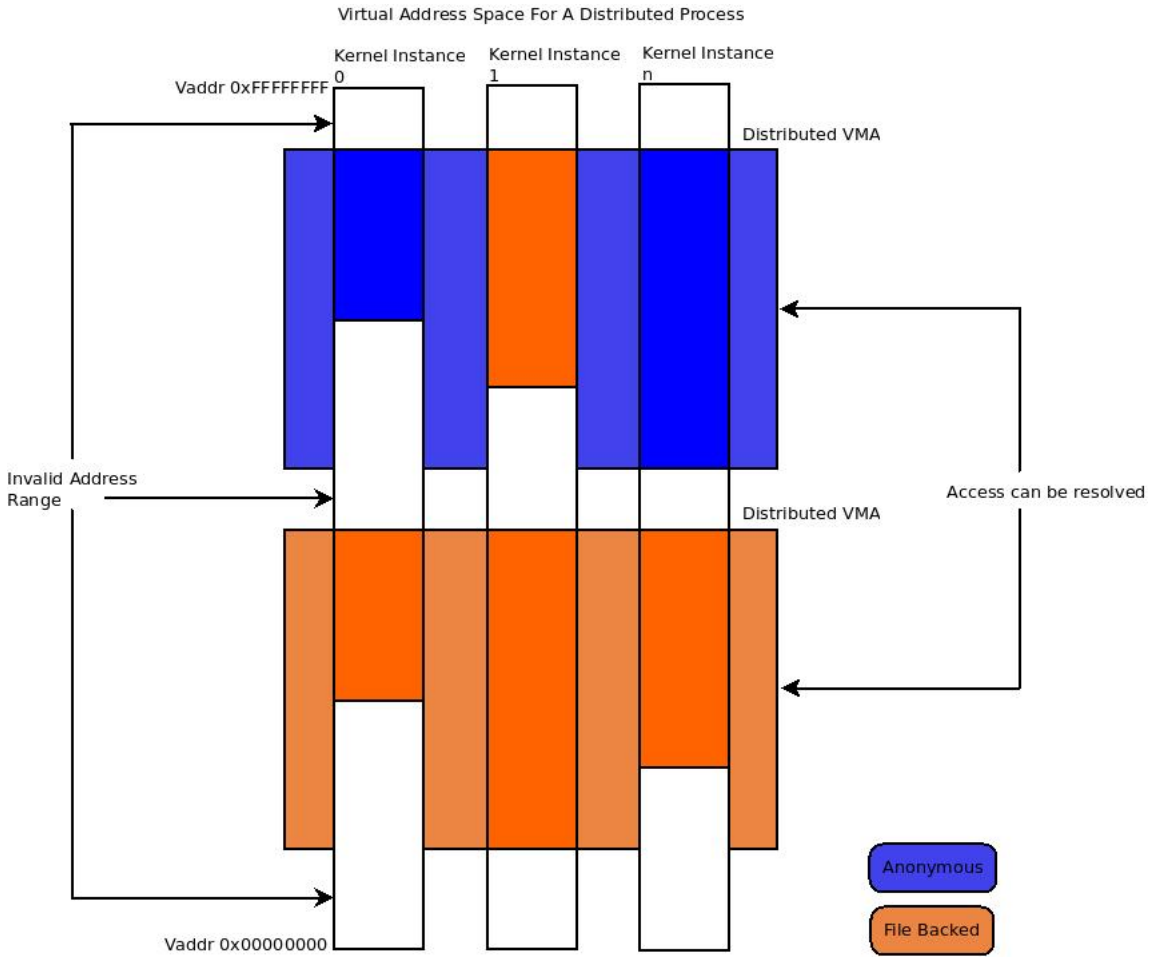

Figure 5.7: An inconsistent distributed VMA in a distributed address space. In light color the full VMA address range, in solid color the VMA mappings known by each kernel instance.

Finally, two mappings cannot be consistent with one another if one is a special mapping, and the other is not. COW pages and the zero page are examples of special pages. This type of inconsistency causes particularly difficult errors to identify. To illustrate this inconsistency, consider the case where a mapping on one kernel instance is COW and the corresponding mapping on another kernel instance is not COW. Assume then that the COW page is accessed and the COW page must now be broken. When this occurs, the kernel instance hosting the COW page will break the COW page, selecting a new physical page to map to that virtual address and copying the contents of the COW page to it. Now, the two kernel instances

are hosting two corresponding mappings that are supposed to be consistent, but instead map different physical addresses to the same virtual address. This creates the first type of inconsistency identified above.

In summary, for two mappings for the same distributed thread group on two different kernel instances to be consistent with one another, the following must be true:

- Either one kernel instance is missing the mapping entirely, or
- All of the following aspects of both mappings must be the same -
  - Virtual address
  - Physical address (Has to be the same or not present on one kernel instance)
  - Protection bits
  - Backing source
    - Backing file (path, and file offset must match), or
    - Anonymous
  - Special page status (COW, zero page, normal page)

5.7.2 Up-Front Address Space Migration

This section describes an early and naive mechanism that was built for migrating a task's address space. It was subsequently obsoleted in favor of the on-demand address space migration mechanism described below. For completeness, the up-front address space migration is described here.

Using this methodology, when a task migrates, its entire address space is migrated at the same time - prior to giving it execution on the remote kernel instance. This procedure involves a page table walk through every page mapping associated with every entry in the VMA list. For each VMA, a message is sent completely describing it. The VMA is then walked, and a message is sent describing every PTE associated with that VMA. The kernel instance receiving these messages stores them, and when it creates the new task, it imports the stored mappings. For every VMA message received, it installs an identical VMA by invoking the do_mmap function. For every PTE message received, it installs an identical PTE by invoking the remap_pfn_range function. Once all VMAs and PTEs have been installed, the address space has been reconstructed and the migrated task is allowed to begin execution.

The up-front address space migration mechanism is not suitable for deployment due to severe performance issues. Some tasks have very large address spaces. Migrating those large address spaces is costly. In many cases, the entire address space is not used after migration, so much of the time spent migrating that address space was wasted. Additionally, this address space

migration mechanism does not support the execution of threads on different kernel instances with evolving address spaces, because it does not consider changes that threads may make to the distributed address space.

5.7.3 On-Demand Address Space Migration

In contrast with up-front address space migration, on-demand address space migration does not migrate mappings during initial task migration. Instead, mappings are migrated as they are needed - as the migrated task executes. This reduces initial migration overhead. Depending on the workload, it can yield significant performance benefits when migrating tasks, especially those with large address spaces. This benefit comes from the fact that the cost of mapping migration is paid only when absolutely necessary, and will be most pronounced in workloads in which migrated threads access only small portions of the address space.

While up-front address space migration is useful for process migration, on-demand address space migration is suitable for use with both distributed processes and threads. The case of threads within the same thread group executing concurrently on different kernel instances introduces interesting problems with respect to maintaining a consistent address space. Any thread group member can at any time remove, change, or add mappings. On-demand address space migration addresses the issue of resolving mappings that are added by remote thread group members after task migration. Mapping removal and modification are also handled in Popcorn Linux, and are described in subsequent sections.

As a user-space program executes, it routinely attempts to access memory for which no virtual to physical address mapping exists. In that case, the MMU will not be able to resolve the mapping, and a fault will occur. In SMP Linux, the kernels fault handler looks up the VMA that corresponds to the faulting address, reserves a page of physical memory for use, and adds a PTE into the tasks page table, mapping the newly reserved page to the faulting virtual page. In the case that a VMA cannot be found, a segmentation fault is reported and the process is killed. This mechanism has been altered in Popcorn Linux to add a step at the beginning of the fault handler that attempts to retrieve existing mappings from remote thread group members.

When a page fault occurs, a query is sent to all booted kernel instances specifying 1) the distributed thread group identity, and 2) the faulting virtual address. The remote kernel instances receive this query, and search for any tasks that they may be currently hosting that are members of the faulting task's distributed thread group. If one is found, its mm_struct is used to resolve the mapping. If a distributed thread group member is not found, it is possible that there was at one time thread group members running there, but they have all since exited. When the last thread group member on a kernel instance exits, but remote thread group members are still executing, the exiting task's mm_struct data structure is saved. That saved mm_struct is preserved until the kernel instance receives notification

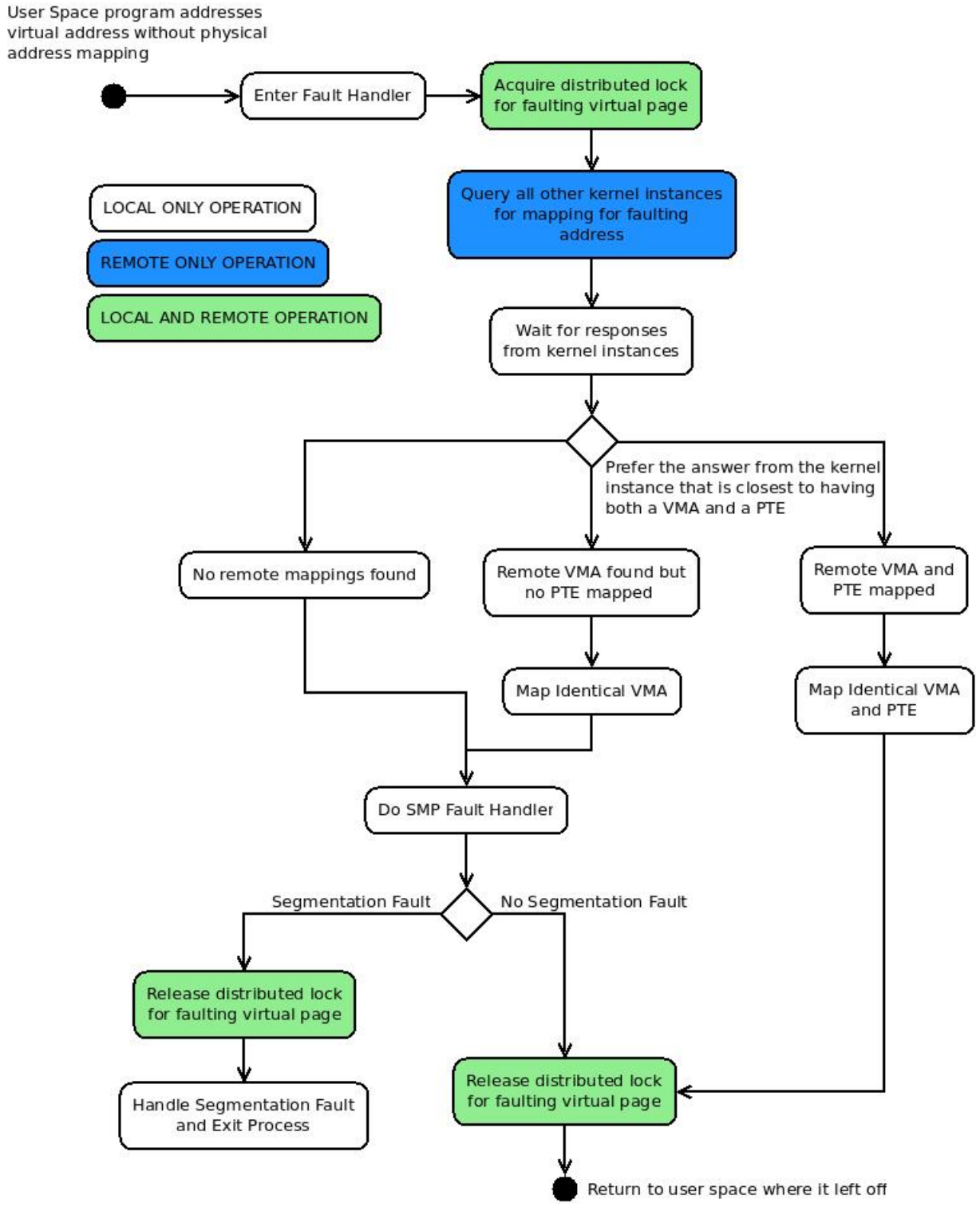

Figure 5.8: On-demand address space migration procedure

that all members of that distributed thread group have exited. When a mapping query is received, and no existing thread group members are found, the list of saved mm_struct's is searched for a mm_struct for that distributed thread group. If one is found, that mm_struct is used to resolve the mapping query. If, after searching existing tasks and saved memory maps, no mm_struct 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. In some cases, a VMA will be found, but no corresponding PTE will be mapped. In that case, a response is sent that fully describes the VMA, but no physical address is communicated.

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, 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. This breaks consistency. It was decided to avoid this situation and pay the performance penalty associated with breaking the COW before the mapping is migrated.

Every running kernel instance responds to mapping requests. The requesting kernel instance, on which the fault occurred, receives responses from every kernel instance and must apply a set of rules to correctly select the best of those responses for use. Some kernel instances will respond with messages that indicate that no mapping exists. Other kernel instances will respond with messages indicating that a valid VMA exists, but no physical page has been allocated to the faulting virtual address. This might be the case if a thread group member on the responding kernel instance invoked the mmap syscall to add to its address space, but has not yet accessed that space. Yet other kernel instances will respond with messages indicating a complete and specific virtual to physical page mapping. 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 mapping exists for the faulting address.

If no remote mapping is found, the normal SMP Linux fault handling mechanism is invoked. If instead, it was found that the VMA exists, but no physical page has been allocated to it, then an identical VMA is installed in the faulting tasks memory map. Once that is done, the normal SMP Linux fault handler is invoked. The normal SMP Linux fault handler will map physical memory to the faulting address. If instead, both a VMA and a physical page mapping are reported to exist, an identical mapping is installed in the faulting tasks memory map. This involves first installing a corresponding VMA if one does not already exist. The physical to virtual address mapping is then installed in the tasks page tables. Unlike the other cases, both a VMA and PTE are assured to now exist, so the normal SMP Linux fault mechanism does not need to be invoked.

Now that the modified fault handler has completed, either the faulting virtual address has a valid corresponding physical address, or a segmentation fault has occurred. If a valid physical address has been mapped, execution is restored to user-space where it left off. When it resumes execution, it will immediately attempt to access the faulting address and succeed.

5.7.4 Concurrent Mapping Retrieval

On-demand address space migration creates a number of concurrency challenges due to mapping retrieval interleavings between kernel instances. Suppose two tasks in the same distributed thread group running in two kernel instances fault on the same virtual address at the same time. Suppose further that no distributed task group members in the system have mappings for that virtual address. In that case, both kernel instances will query all other kernel instances for a mapping. Because the mapping does not yet exist, neither will receive any positive responses. Each will then default to the normal Linux fault-handling mechanism, wherein a new physical page is assigned to the faulting virtual page. Each kernel instance will select a physical page from its own range of the address space. This results in a distributed address space in which there are two physical pages mapped to the same virtual page. This is an inconsistent state. In order to remove this failure mode, some notion of atomicity must be introduced. Care must be taken when introducing mechanisms for atomicity in performance critical distributed systems. In order to provide relatively fast critical regions for fault mapping, Lamport's distributed mutual exclusion algorithm (LDMEA) is used.

LDMEA calls for each party (P) to have a queue (Q) for each resource that is being guarded. When P_x needs access to R_i, it creates a time-stamp, T_a, and sends a request to all other parties containing the time-stamp, and the resource required REQUEST(T_a,R_i). It also places REQUEST(T_a,R_i) on its request queue corresponding to the resource, $Q(x)_{R_i}$. When remote parties (P_y) receive message REQUEST(T_a,R_i), they place those requests on their queue for that resource, $Q(y)_{R_i}$ ordered by time-stamp. Once P_x sends its REQUEST message, it waits for REQUEST(T_a,R_i) to get to the front of $Q(x)_{R_i}$. A party has gained critical access to resource R_i once its request is at the front of the queue. When a party is finished with its critical region, it sends a message to all other parties to signify the resource release, containing the original time-stamp and resource RELEASE(T_a,R_i). When a party receives a release message, it removes the queue entry corresponding to REQUEST(T_a,R_i) from its resource queue $Q(y)_{R_i}$.

To reduce lock contention, a Lamport queue is created for every (distributed thread group/-faulting virtual) page pair. This ensures that Popcorn can concurrently resolve mappings for tasks in different thread groups, or tasks that are in the same distributed thread group but are faulting on different virtual pages. In order to minimize the memory requirements for this part of Popcorn, queues are created dynamically as they are needed, and are destroyed

when they are no longer needed. Each queue is specific to a distributed thread group / virtual page tuple (TG_a, $VPage_1$). A new queue is created if either the faulting kernel instance does not have a queue for a given (TG_a, $VPage_1$) and a queue does not already exist, or if it receives a REQUEST for a virtual page's lock and a queue does not already exist. A queue continues to exist as long as it contains REQUEST entries. Once the last REQUEST entry is evicted from a queue, that queue is garbage collected.

LDMEA requires that all nodes share a common notion of time. Unfortunately, the time stamp counter (TSC) that is integrated into the x86 fabric is not guarantee to be synchronized across cores. In order to provide synchronized 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. The fetch and add instruction is an atomic memory operation, and therefore thread safe across kernel instances.

5.7.5 Virtual Memory Area Modification

Concurrency During Modifications

The following sections describe methods of maintaining a consistent address space through address space write, modification, and removal operations. Those operations have the potential to create errors during concurrent execution, unless mitigations are implemented. In order to provide strong assurances that concurrent virtual memory area modifications do not create address space inconsistencies, each of the operations described below are wrapped in distributed locks. The locks are implemented with Lamport's distributed mutual exclusion algorithm, also described above. When a munmap, mremap, mprotect, or mmap operation is executed against a region of memory, a lock is secured. During these operations, a lock is acquired on the entire virtual address space. The overhead associated with creating Lamport queues for every virtual page in the address space, as is done during mapping retrieval, is prohibitive. To remove this performance hit, an alternate method was built which still locks the entire address space, but without creating queues for every virtual page. We call this lock a *heavy* lock. A single heavy Lamport queue exists on every kernel instance for every distributed process. This heavy lock represents the Lamport locks for all virtual pages in a distributed address space for which a fine-grained per page Lamport queue does not already exist. A heavy lock behaves exactly the same as one of the fine-grained per page locks that are used for page retrieval, 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, as well. Also, 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 64 bit address space, only the per virtual page queue creation is deferred. 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 LDMEA implementation for mapping retrieval.

The Lamport lock name-space used to guard virtual memory area modification is shared with the Lamport lock name-space that is used to guard against concurrent mapping retrieval, as described above. Because of that, it is assured that not only will concurrent operations that modify the address space be prohibited, but also those operations will never occur as a mapping is being migrated. This covers the corner case where a mapping migration occurs in a kernel instance while that same mapping is removed or modified in another kernel instance, which would constitute a race condition. By adding heavy lock entries into all per page queues, it is assured that no task attempting to acquire a lock on a per page queue is successful until all heavy atomic operations that have higher time precedence have completed.

Each of the operations that effects the address space has been modified as follows:

```
// Modified procedure for munmap, mprotect, mremap,
// mmap
memory_map_modifying_function(address, length):

    // Acquire a lock against the
    // entire virtual address range.
    acquire_lamport_lock_heavy();

    // It is now safe to modify the region of interest,
    // as it is now guaranteed that no kernel instance
    // (even another task on this kernel instance) can
    // operate on the address space.

    // In most cases, this effects both local and
    // remote memory map information in order to
    // maintain consistency.
    do_memory_map_modifying_function(address, length)

    // Release the lock taken against
    // the entire virtual address range.
    release_lamport_lock_heavy();
```

Handling Page Mapping Removal

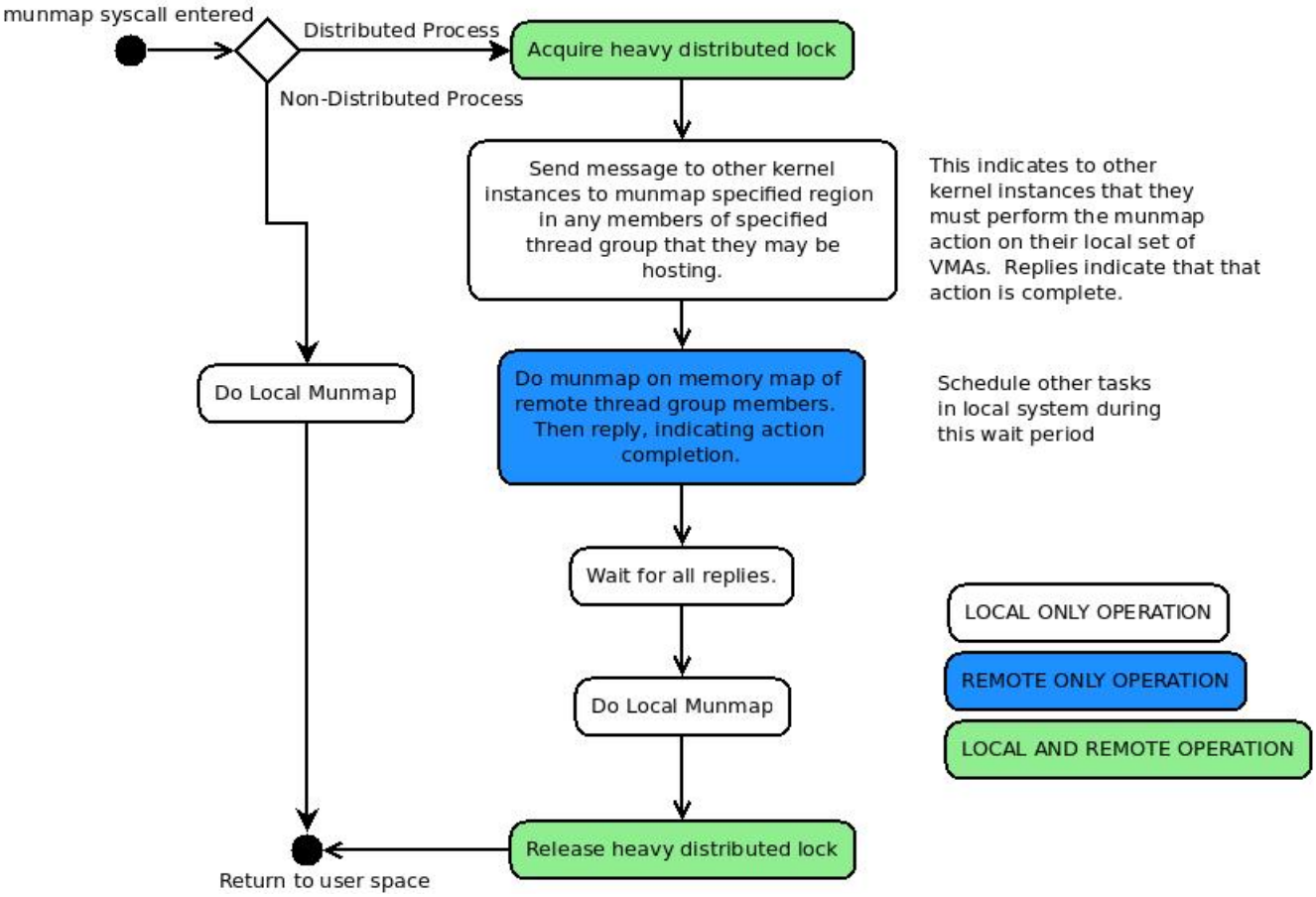

Figure 5.9: Munmap procedure

The munmap syscall allows a program to remove all or part of a VMA from its address space. In Popcorn Linux, to maintain address space consistency, this action must be performed on all members of the distributed thread group. The munmap implementation has been augmented to perform a synchronous distributed munmap. A message is sent to all remote kernel instances indicating that they must perform a munmap on the specified distributed thread group for the specified portion of its address space. After each remote kernel instance performs the munmap action, it responds acknowledging the request. The requesting task waits in kernel mode for all responses to arrive before continuing execution. While it waits, the kernel schedules other tasks to execute. By waiting for all munmap acknowledgments to come in, it is ensured that the task that called munmap will not resume until the entire distributed address space reflects the changes that it requested. Once all remote kernel instances have completed the required unmap operation, the local unmap is done, and control is returned to user space. See Figure 5.9 for a flow chart of the Popcorn unmap procedure.

Handling Page Mapping Modification

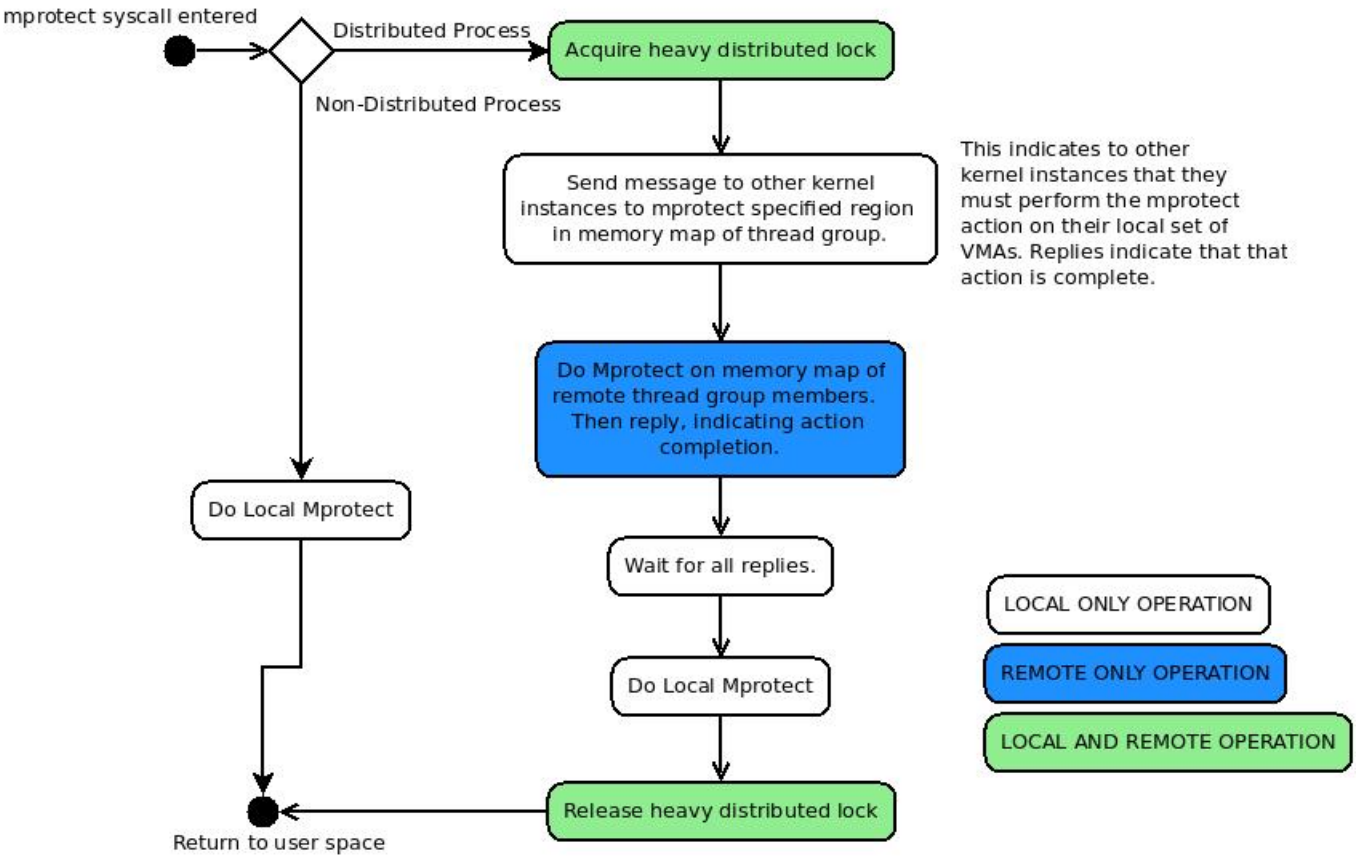

Figure 5.10: Mprotect procedure

The mprotect syscall allows a program to modify the protection applied to existing VMAs in its address space. The user-space program must specify the address at which to start the modification, and the size of the memory region, as well as the new protection mode for the region. When an mprotect syscall is invoked, Popcorn distributes the mprotect call across all kernel instances. It sends a message to all remote kernel instances specifying the address range, new protection mode, and the distributed thread group identification information for the thread group that should undergo the mprotect action. When a remote kernel instance receives this message, it looks up any mm_struct's associated with the specified distributed thread group, then applies the mprotect action to that memory map. Once the remote kernel instance has completed the mprotect operation, it acknowledges the mprotect request. The kernel instance that requested the mprotect action waits to receive acknowledgments from all available kernel instances. Once it has received responses from all kernel instances, it performs the local mprotect then resumes execution in the syscall and returns to user space. This mechanism ensures that execution is not given to a task that has called mprotect until

the mprotect action has been applied to the entire distributed address space. See Figure 5.10 for a flow chart of the Popcorn mprotect procedure.

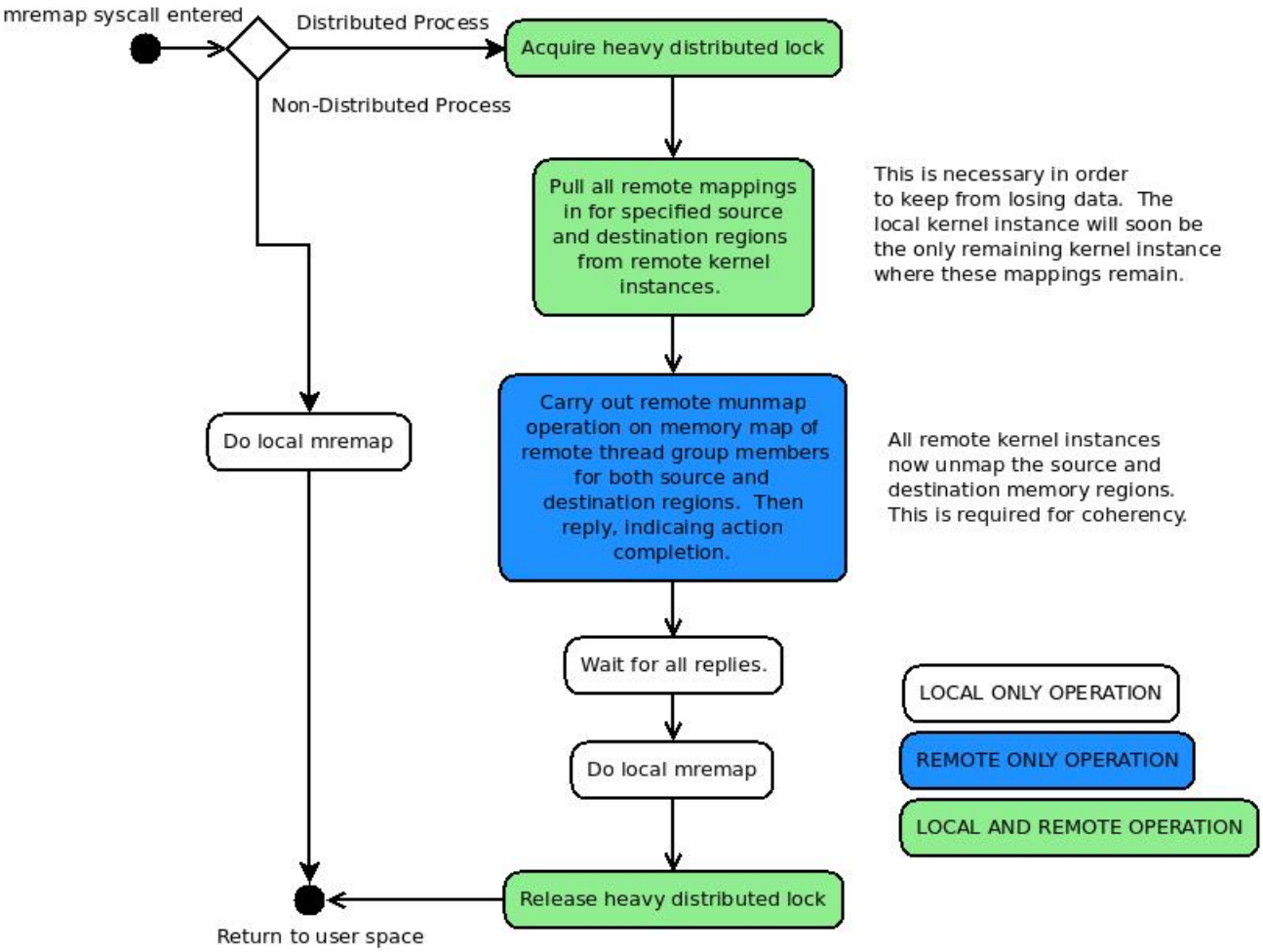

Figure 5.11: Mremap procedure

The mremap syscall is another mechanism through which the address space may change. This system call allows for a mapped virtual memory range to be expanded, shrunk, or moved to another range of virtual addresses. The mremap syscall takes as parameters a description of the source range, including a source address and the ranges size, the desired new size, and a set of flags that tell the kernel how to treat this remap call. There is a flag to tell the kernel whether or not it may move the region to another virtual address if necessary, and another to tell the kernel that it must move the address range to a specific new address. In the case that a specific destination virtual address is supplied, a fifth argument is taken which holds that destination address. When the destination address range is already occupied, those conflicting mappings are unmapped prior to moving the old mappings into

that new region. Popcorn modifies this procedure. Popcorn must take special action to ensure address space consistency when the changes resulting from mremap calls occur. The first thing that Popcorn does is pull in all of the remote mappings for the region that is about to be remapped. This action preserves the contents of those mappings, as soon the local kernel instance will be the only kernel instance maintaining those mappings. Popcorn then issues a message to all remote kernel instances indicating that they must unmap both the old region and the new region from any memory maps that they are hosting for the thread group. Once the remote unmap operation is complete, the kernel instance that initiated the remap is the only remaining kernel instance that still has the VMA and PTE entries associated with the affected memory regions. This maintains consistency by preventing remote mappings from conflicting with the ones that will soon exist locally. Once all remote actions have completed, the local mremap is carried out. See Figure 5.11 for a flow chart of the Popcorn mremap procedure.

Safe Memory Mapping Support

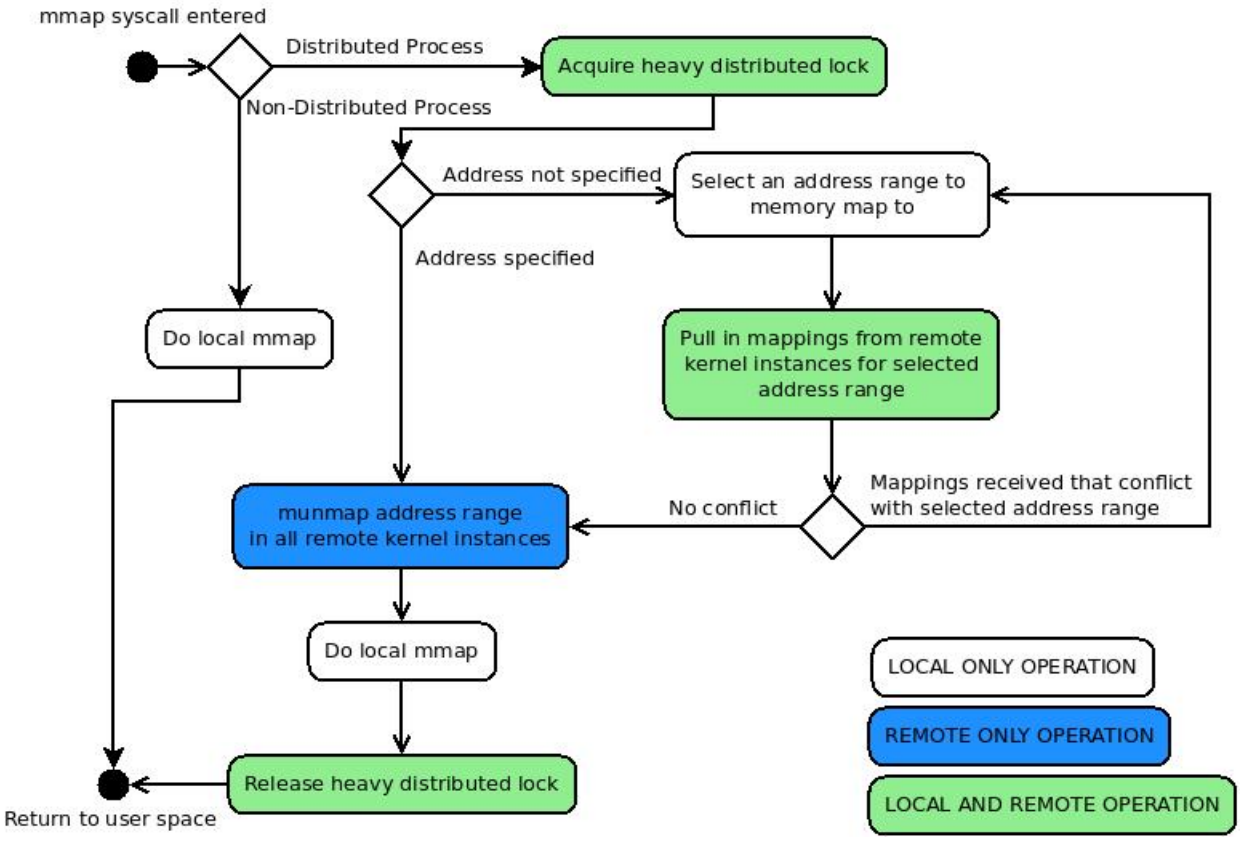

Figure 5.12: Mmap procedure

A process can ask the kernel to create new virtual memory areas by invoking the mmap syscall. The process passes arguments to that syscall to configure how the kernel will map the new memory. For example, it can specify a path to a file, and the kernel will memory map the contents of that file to the new virtual memory area. The program can also specify no file, in which case the kernel will map anonymous physical pages to the new virtual address range. A program has the option of specifying a virtual address at which to place the new virtual memory area. In this case, any pre-existing virtual memory areas in that region will be unmapped to make room for the new one. A program might, in some cases specify a virtual address to map to, or it might let the kernel decide. Some of these capabilities represent corner cases from a distributed address space consistency perspective.

When a program specifies a virtual address range to map, and that address range overlaps an existing virtual address range, the old overlapping areas will be unmapped. However, a given Popcorn kernel instance might not know about all valid remote virtual address ranges. Because it does not know about all mappings, it cannot be trusted to do distributed munmap operations on the conflicting address ranges since it could miss virtual memory areas that reside only on remote kernel instances. Popcorn deals with this case by carrying out a distributed munmap operation on the entire target address range whenever a new address range is created. This ensures that no remote kernel instances will contain conflicting mappings.

An interesting corner case occurs when two or more kernel instances receive instruction from a program to memory map a new virtual memory area without a specific address supplied. In this case, each kernel instance must select a virtual address range to memory map to. It is possible in this case that both kernel instances will select the same address range. This creates an inconsistent state, since two local VMAs will now exist in the same region of the distributed address space. In order to deal with this corner case, Popcorn takes special action when a memory map operation is invoked and an address is not specified. After selecting the address to mmap, the kernel instances fetches all remote mappings for the selected address range. If any are found to conflict, a new address is selected. This process is repeated until an unused address range is selected. This operation is done atomically using the same Lamport's distributed mutual exclusion algorithm described above. This ensures that these fetch actions do not conflict with fetches triggered by either local or remote page faults. See Figure 5.12 for a flow chart of the Popcorn mmap procedure.

5.7.6 Mapping Prefetch

There is non-trivial overhead associated with mapping retrieval. Because of this, mapping prefetch was implemented in an attempt to reduce the number of mapping retrievals that must be done to satisfy a workload. The message that carries mapping responses was extended to include a configurable number of prefetch slots. Each prefetch slot contains:

- Starting virtual address (vaddr_start)
- Starting physical address (paddr_start)
- Size of contiguously mapped region (sz)

Each prefetch slot describes sz / PAGE_SIZE pages of memory in which the virtual addresses, starting at vaddr_start, are mapped to physical pages that start at paddr_start, and both increase contiguously. By describing mappings in this way, large quantities of mappings can be described in a compressed format.

When the mapping retrieval request comes in, the receiving kernel instances searches for a mapping for the faulting address. If a VMA is present for the mapping, the contiguous mapping regions including and surrounding the faulting address in the same VMA are extracted from the memory map and placed in prefetch slots. 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 VMA to fill the slots, the remaining slots are left empty.

When mapping responses are received on the kernel instance that initiated the mapping retrieval, it installs the prefetched mappings. The mapping that was originally asked for is included in a prefetch slot.

This work is done while the Lamport lock is held for the faulting address. It is not necessary to lock the prefetched pages because those pages are guaranteed not to change during this operation. This guarantee comes from the fact that mmap, mprotect, munmap, and mremap operations can never occur during a mapping retrieval.

Chapter 6

Results

Five macro benchmarks were used to demonstrate the correctness of the Popcorn task migration and address space consistency algorithms, as well as to show performance under different types of workloads. The results are used to draw conclusions about which aspects of Popcorn task migration and address space consistency perform well, and which can be improved. For each of the macro benchmarks that were measured, the results are preceded by a detailed description of how the workload functions: what types of page fetches it performs, how many task migrations it performs, etc. This information is critical to the analysis, as different types of workload behaviors result in different overheads.

Results for some of the benchmarks are also presented for Barrelfish. The Barrelfish source which was taken from the Barrelfish repository. The change set that was used was 1298:12e657e0ed48 from hg.barrelfish.org, which was committed on March 22, 2013. The Barrelfish team was consulted to understand the overhead, but there was not enough time to analyze the source and fully understand the overheads before this material was due for presentation.

All Popcorn Linux, SMP Linux, and Barrelfish results were collected on a homogeneous x86 64bit, 64 core, 4x AMD Opteron 6274 running at 1.4GHz with 128 GB RAM.

In order to understand the cause of the differences between Popcorn and SMP Linux performance, measurements from the *perf* tool are provided for each workload which show the overhead contributions from the most significant sources of overhead. Because a distributed *perf* tool does not exist to provide this kind of measurement in Popcorn across kernel instances, only the main thread was measured for both Popcorn and SMP Linux. This ensures that fair comparisons are made between the two platforms. Note that this does not reduce the trustworthiness of the analysis because the applications are structured such that the main thread acts as a master and incurs most of the costs, as is typical of OpenMP applications. The *perf* tool is configured to sample at a rate of 4KHz. These measurements are not intended to be used to predict benchmark results, because they capture only the main thread. Rather, these measurements provide insight into which overhead components are

removed by Popcorn from SMP, and which are added by Popcorn on top of SMP.

All benchmark measurements are provided in Time Stamp Counter (TSC) units except where otherwise noted. A TSC corresponds to a single clock cycle. Results are presented in this unit of time to remove CPU frequency from the measurements.

Each of the benchmark workloads was adapted to Popcorn to make use of the custom threading library that is used in the Popcorn user space, cthread. Cthread was developed by Antonio Barbalace to replace the pthread library. Pthread is not currently supported in Popcorn due to the lack of distributed futex functionality.

6.1 Microbenchmarks

6.1.1 Mechanism Costs

A microbenchmark application was written to measure the amount of time needed to do task migration, address space migration, and task life cycle maintenance. The amount of time for each operation was measured in kernel mode, then averaged. The measurements are provided below. This microbenchmark application was designed to be lightweight. The numbers are likely to change under heavier load. One operation that will experience significant performance variability as load changes is Lamport lock acquisition. Lamport lock acquisition time will vary from application to application, and execution to execution, since lock acquisition interleavings and contention will vary. Regardless, the microbenchmark numbers, presented in Table 6.1, are useful for providing an understanding of the magnitude of work associated with each operation, and that understanding can then be applied when analyzing the macro benchmark results that are provided in later sections.

Table 6.1: Microbenchmark results

Procedure Measured	Average Time (TSC)
Importing a migrated task, including first mapping retrieval	188902
Distributed task exit	134859
Distributed group exit	6223
Distributed mprotect	40598
Break COW page	18836
Counting remote thread group members	237263
Waiting for Lamport Lock	27777

6.1.2 Task Migration Measurement

To measure task migration times, a microbenchmark application was written which performs two migrations. During each migration, it reads the TSC immediately before, and immediately after the migration, subtracting the two to arrive at the total migration time. The first migration moves the process to a remote kernel instance. The second migration moves the process back to its original kernel instance. The Popcorn migration mechanism is different for moves to new kernel instances than it is for moves to kernel instances on which a task has already executed. The number of active kernel instances was varied to see the effect of added messaging associated with page retrievals that occur prior to the TSC read operation immediately after the migration. The results of these measurements are shown in Figure 6.1, and are compared to similar measurements taken on SMP Linux.

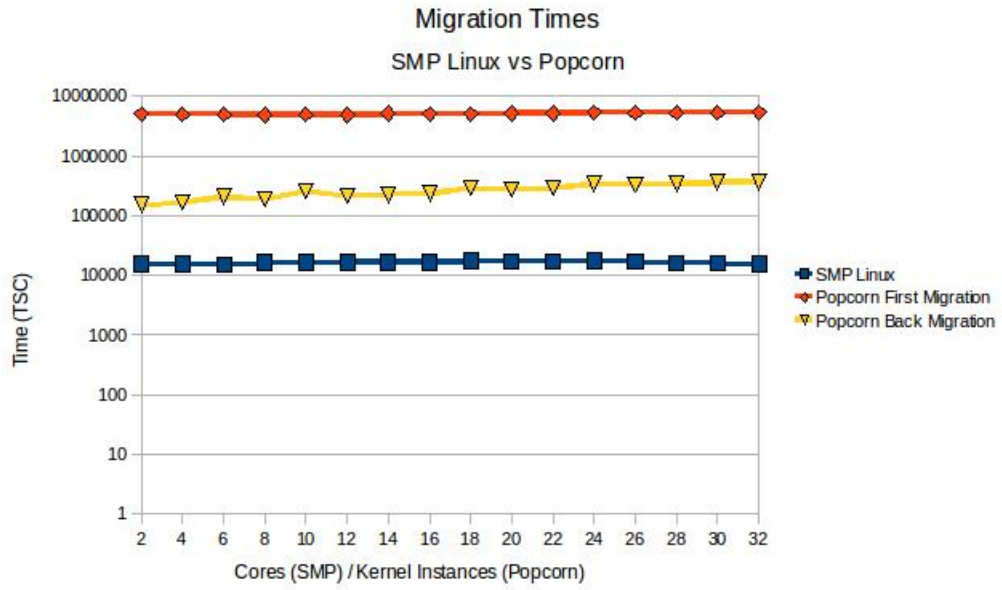

Figure 6.1: Migration time measurements comparing SMP to Popcorn Linux

These measurements show that it takes significantly longer to migrate to a new kernel instance than it does to migrate back to a previous kernel instance. This is due to the optimizations that were discussed in 5.6.2. SMP Linux clearly outperforms Popcorn task migrations. This is due to the fact that it does not need to use any messaging to communicate the task's state, or migrate any pages.

6.2 IS-POMP

The is-pomp workload is a parallel integer sorting benchmark designed to test random memory access. Is-pomp is a Popcorn adaptation of the is-omp workload designed by NASA in the NAS Parallel Benchmark suite[2]. The following section describes the is-pomp workload in order to give the reader an understanding of what aspects of Popcorn this benchmark exercises most. Once the workload profile is explained, the is-pomp Popcorn results are presented. Popcorn performance is also contrasted with the performance of the same workload on both SMP Linux, and Barrelfish OS.

6.2.1 IS-POMP Workload Profile

The is-pomp workload can be run across as many kernel instances as are present. When the is-pomp program is invoked, the number of kernel instances to use are specified as a command-line argument. The is-pomp workload performs two task migrations for every kernel instances that is used beyond the first kernel instance. If only one kernel instance is used, no task migrations occur, and all processing is done in the kernel instance on which the program was invoked. The number of migrations for an invocation of is-pomp involving N kernel instances is therefore 2(N-1).

Figure 6.2 shows the frequency of various significant events as the workload is executed on a varying number of kernel instances. The event types were chosen to show overhead specific to features added to the Linux kernel by Popcorn for task migration and address space maintenance. To better understand the events shown in Figure 6.2 and the others in this section, please see 5.7. The is-pomp workload Popcorn overhead is dominated by fault-triggered page fetches which resulted in the retrieval of pages that are file-backed with no PTEs already mapped. This event type is a result of a large file-back memory map operation taking place in user space. The frequency of this event type increases linearly as the number of kernel instances involved in the program's execution increases.

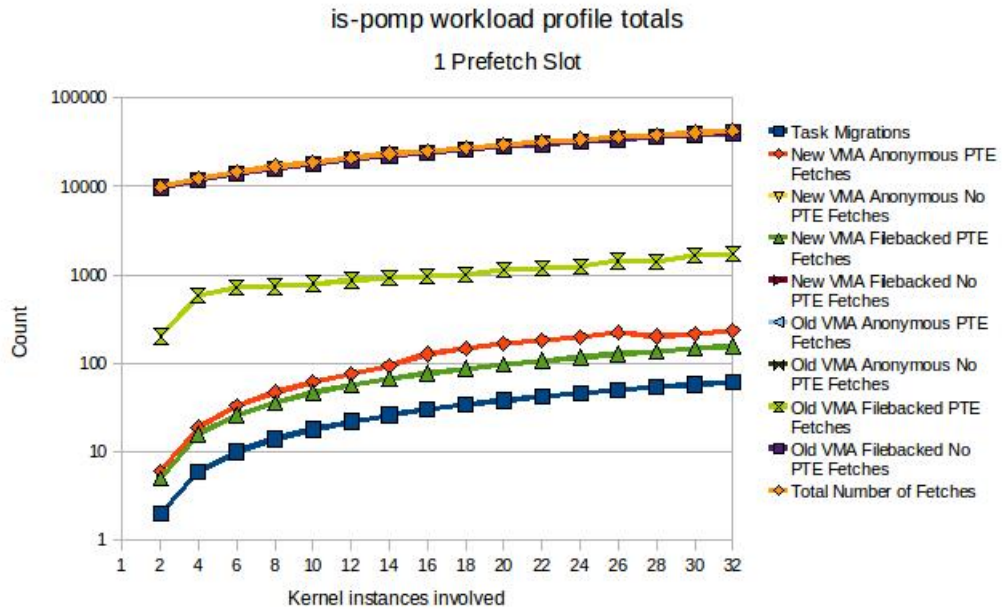

Figure 6.2: IS Workload Event Totals With 1 Prefetch Slot

Figure 6.3 shows details on how the totals shown above break down per kernel instance for the specific case where 16 kernel instances are involved. 16 kernel instances was picked because there are enough kernel instances to show trends. This graph shows that the number of mapping retrievals resulting in file backed mappings with no mapped PTE are distributed evenly across all kernel instances except kernel instance 0, on which the program was initiated. This even distribution is the reason why the number of fetches of that type increase linearly as the number of kernel instances increase, as shown above.

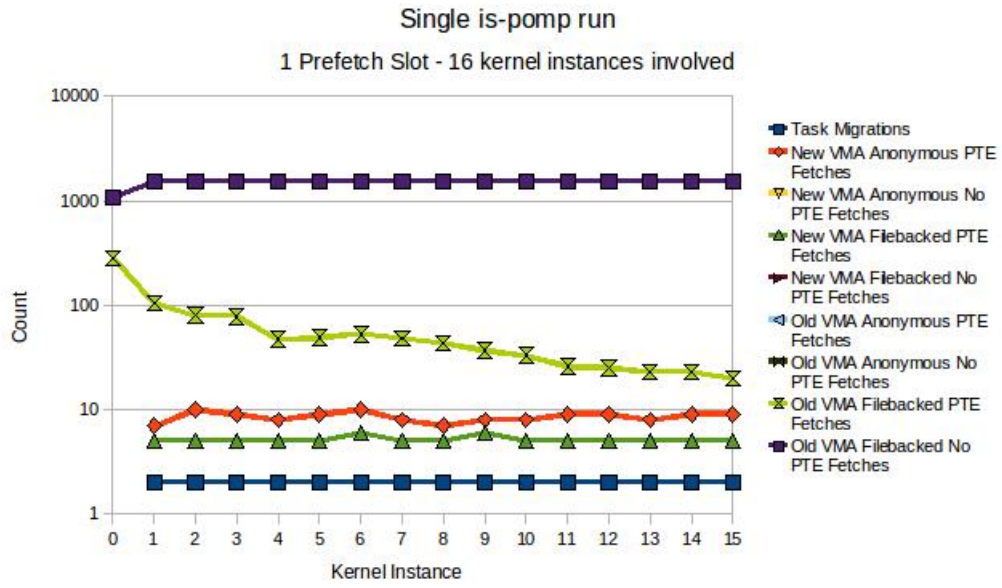

Figure 6.3: IS Workload profile with 16 kernel instances with 1 prefetch slot

Table 6.2: IS-POMP fault processing times by kernel instance with 1 prefetch slot

CPU	Count	Max	Min	Average
0	1366	25559213	62987	437269
1	1653	25625055	77537	433763
2	1631	25626566	79990	527377
3	1628	25580568	78731	572261
4	1596	25545524	79659	573213
5	1599	25550431	79642	600962
6	1607	25614348	69762	546000
7	1598	25583859	93462	535940
8	1591	25610945	75730	539804
9	1589	25587815	89118	590323
10	1582	26388953	100556	571942
11	1576	21782294	68154	569573
12	1575	26367835	98208	566079
13	1572	25614954	95966	539351
14	1573	25622960	92131	555915
15	1570	22835949	79840	534643

The following figures show how this workload responds to increasing the amount of prefetch

that is performed during each mapping retrieval. 4 prefetch slot and 8 prefetch slot profiles are shown in these graphs, as opposed to the single prefetch slot that is shown above. Because the is-pomp workload is dominated by mapping retrievals that do not result in successfully matching virtual address to physical addresses, no noticeable workload profile changes are seen between prefetch amounts.

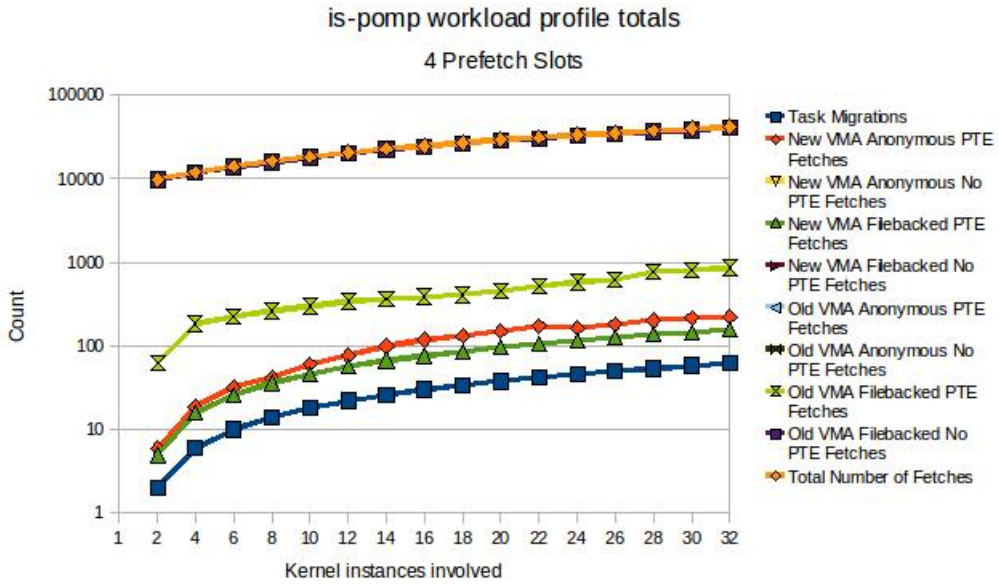

Figure 6.4: IS Workload Event Totals With 4 Prefetch Slots

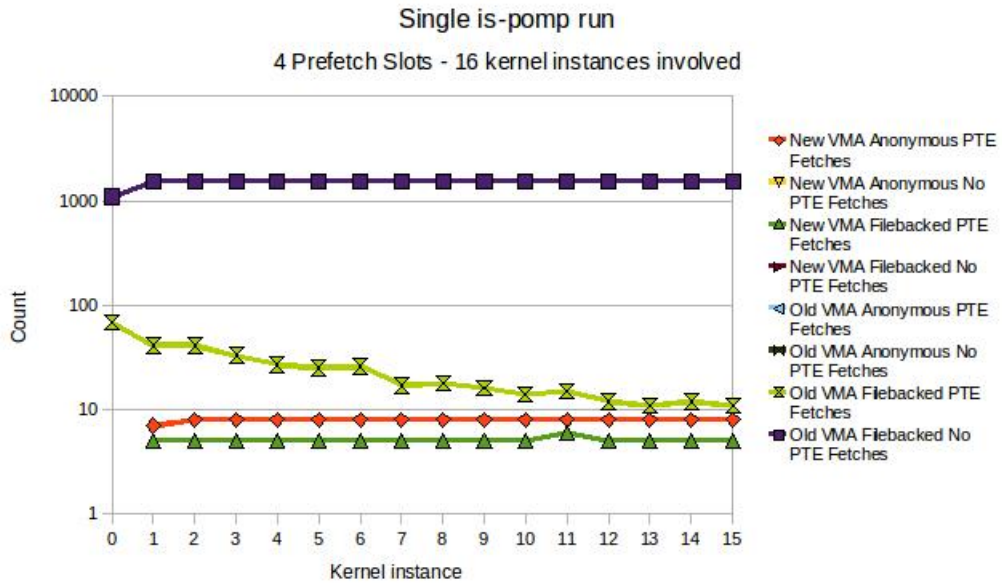

Figure 6.5: IS Workload profile with 16 kernel instances with 4 prefetch slots

While the number of fetches is not affected by increasing amounts of prefetch in this workload, the amount of time needed to do a fetch does increase as the degree of prefetch increases.

Table 6.3: IS-POMP fault processing times by kernel instance with 4 prefetch slots

CPU	Count	Max	Min	Average
0	1154	26624037	61875	618130
1	1589	15476044	90038	500245
2	1590	21908451	82225	640926
3	1582	21790102	107693	553513
4	1576	21738265	80272	552982
5	1574	21787306	106370	543577
6	1575	23985382	77397	555094
7	1567	23973831	93388	573548
8	1567	23976857	74239	591407
9	1565	22158828	91458	593803
10	1563	21886493	95292	543085
11	1567	24505192	83729	592357
12	1561	21979086	98417	591262
13	1560	21796833	86316	503128
14	1561	21840309	105135	522660
15	1560	21847817	75730	556105

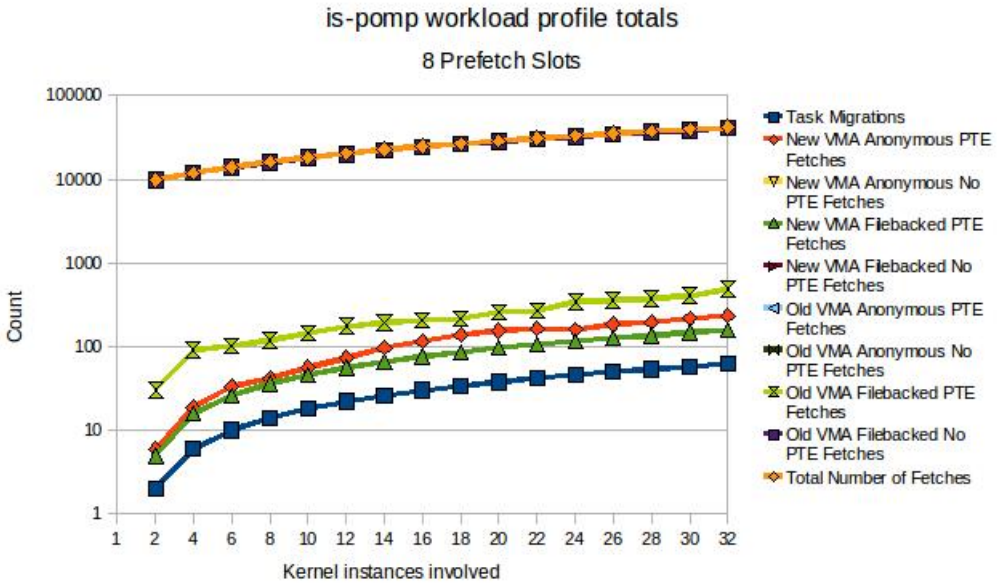

Figure 6.6: IS Workload Event Totals With 8 Prefetch Slots

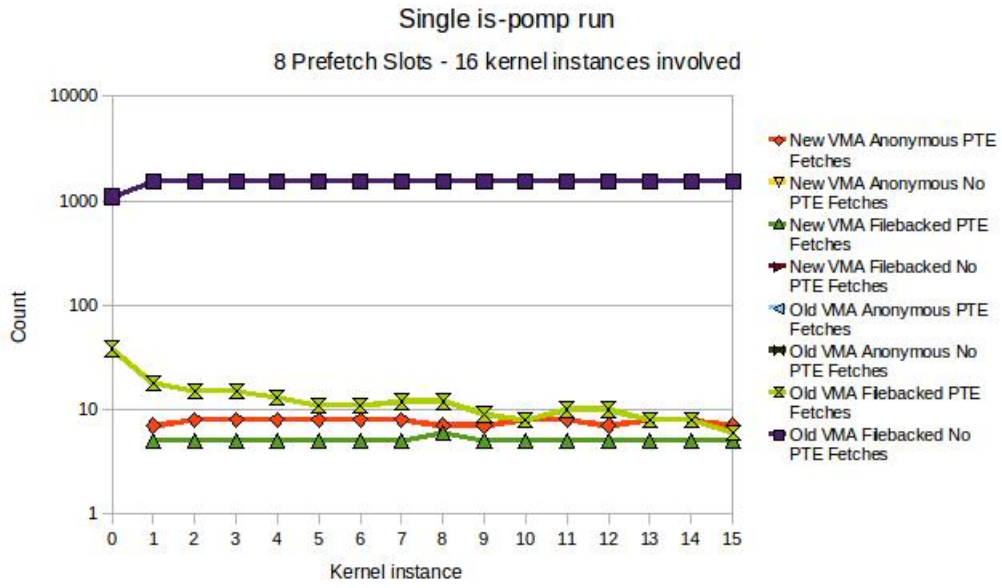

Figure 6.7: IS Workload profile with 16 kernel instances with 8 prefetch slots

Table 6.4: IS-POMP fault processing times by kernel instance with 8 prefetch slots

CPU	Count	Max	Min	Average
0	1124	33653767	62386	639072
1	1566	26737716	96095	532637
2	1564	27018274	105415	628215
3	1564	27169978	84919	611558
4	1562	21847199	93059	646780
5	1560	21926538	101758	592536
6	1560	26794655	81768	593831
7	1561	27126825	104038	615748
8	1563	28965564	74164	701775
9	1558	27109819	73808	751314
10	1557	27201521	83367	633790
11	1559	26670385	85449	660100
12	1558	26732041	90229	620883
13	1557	26739241	101290	540090
14	1557	26722496	76161	705805
15	1554	21923298	80811	568412

In order to understand how much of the Popcorn overhead was due to the Popcorn message

passing layer, the total time spent messaging during a mapping retrieval was measured. This measurement does not include the Lamport lock acquisition and release since that time is variable based on whether or not the lock is already held by another kernel instance, and could drastically skew the result. The message transit times were arrived at by measuring the time before a message was sent, and then again after it arrived. This was done in both directions, once for the request, and once again for the response. By measuring twice, once in each direction, any TSC drift is eliminated from the total. The transit measurement is compared to the total time needed to carry out a mapping fetch to arrive at a messaging overhead. For the average is-pomp execution involving 2 kernels, the messaging overhead is 40%, 38%, and 40% for 1 prefetch slot, 4 prefetch slots, and 8 prefetch slots respectively. This percentage decreases slightly due to the increased overhead associated with looking up mappings, and then installing them at their destination.

Table 6.5: IS-POMP 2 Mapping Retrieval Message Transport Times

	1 Prefetch Slot	4 Prefetch Slots	8 Prefetch Slots
Avg Mapping Fetch Processing (TSC)	42011	43432	42872
Total Message Transit (TSC)	17184	16841	17504
Amount Of Mapping Fetch Processing Waiting For Msg Transit (%)	40.90	38.77	40.82

The perf tool was used to understand which overhead components were most significant in both SMP Linux and Popcorn, and see which ones increased and decreased between the two platforms. These measurements were taken on the main thread of the workload. The greatest kernel-mode overhead contributors are shown graphed below, as well as the remaining kernel-mode overhead, and the breakdown of time spent in user-mode versus kernel-mode.

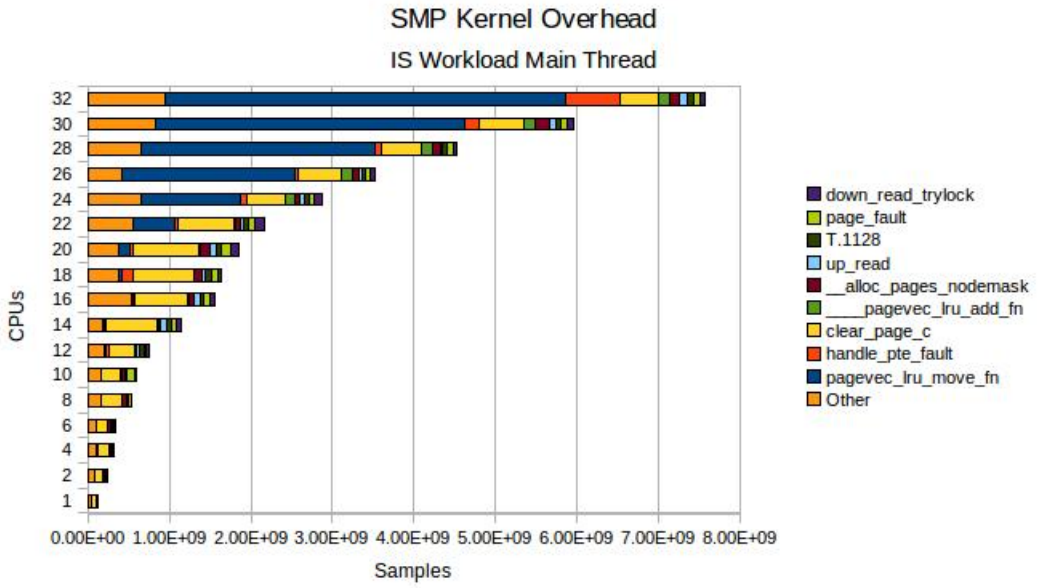

Figure 6.8: Perf Measurement for IS-POMP Workload on SMP Linux

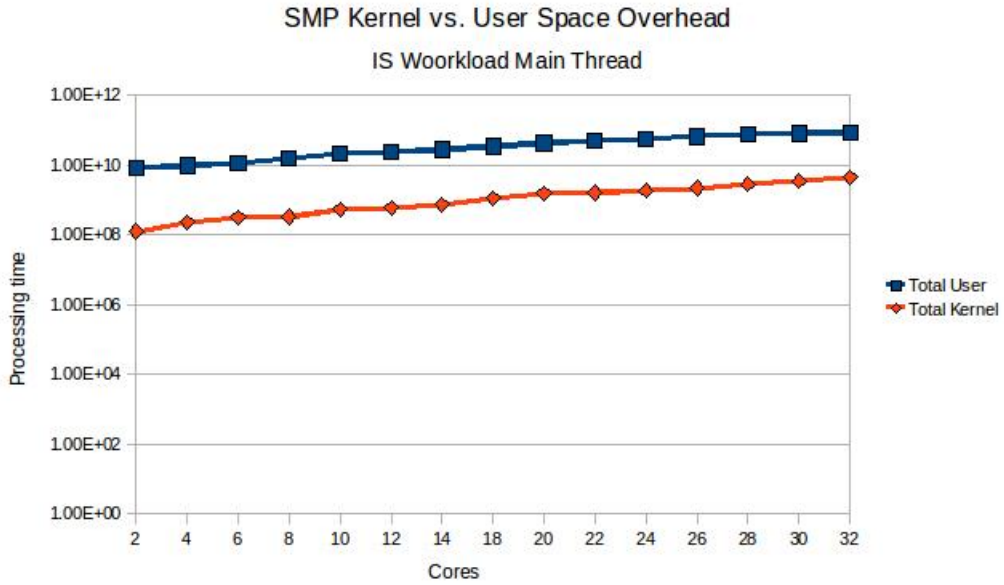

Figure 6.9: Perf Measurement for IS-POMP Workload on SMP Linux - User vs Kernel

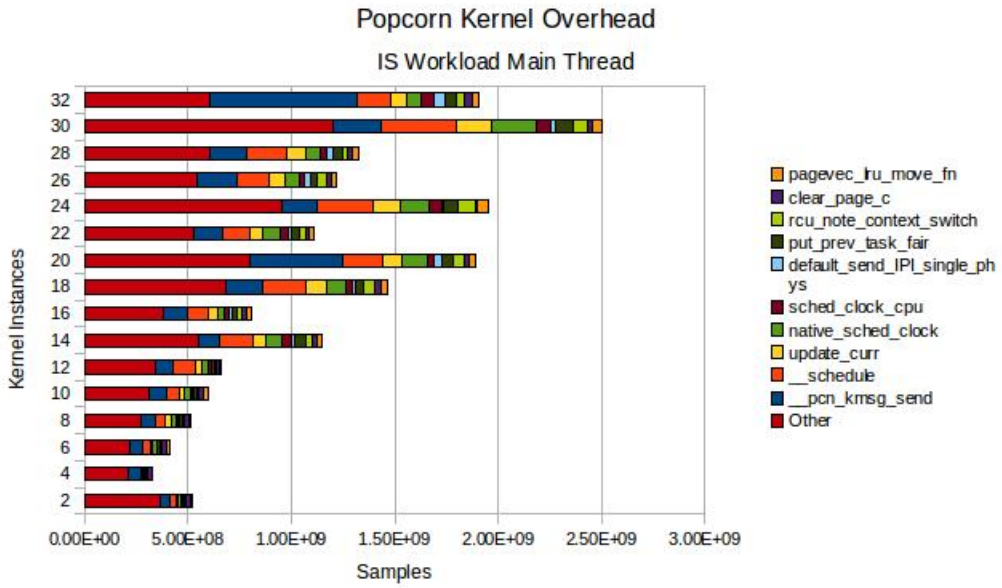

Figure 6.10: Perf Measurement for IS-POMP Workload on Popcorn

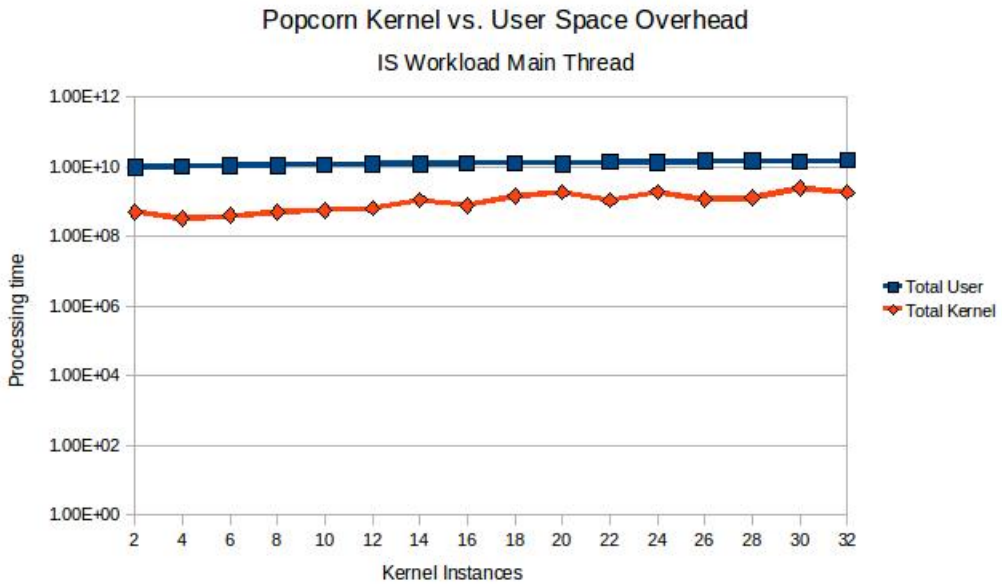

Figure 6.11: Perf Measurement for IS-POMP Workload on Popcorn - User vs Kernel

SMP Kernel overhead is increasingly dominated by time spent in *pagevec_lru _move_fn*. This function is called during the page mapping process. That function creates significant overhead due to contention for a spin lock. As the number of cores contending for that spin

lock increase, contention also increases. The other significant source of overhead is the *clear_page_c* function, which zeros out the contents of a page. This function is also part of the memory management subsystem. Its overhead remains relatively constant and significant as the number of CPUS in use increase. While these sources of overhead are still present in the Popcorn overhead, they are hugely reduced in magnitude due to memory subsystem decoupling and are not nearly as significant. The Popcorn overhead is instead dominated by *__pcn_kmsg _send*. This function handles sending messages in Popcorn, and has already been flagged as a primary source of overhead.

6.2.2 IS-POMP Results

Popcorn Linux outperforms competing projects on the is-pomp workload. The removal of the necessity to access shared memory resources for scheduling and other purposes allows Popcorn to realize performance gains on this workload that outweigh the overhead introduced by Popcorn's task and address space migration and consistency algorithms.

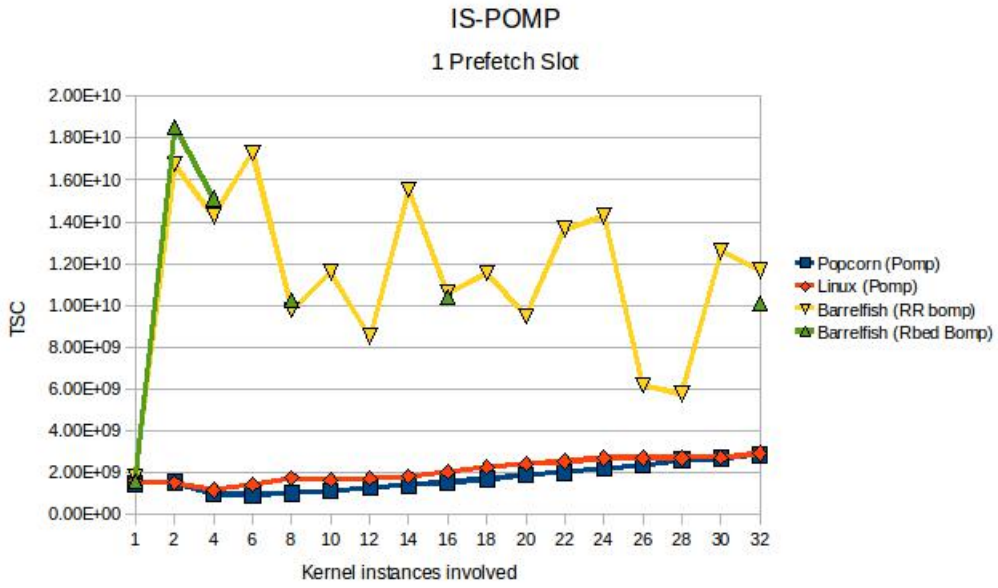

Figure 6.12: is-pomp results varying involved kernel instances with 1 prefetch slot

Prefetch has negligible effect on the is-pomp workload due to the fact that the Popcorn overhead is hugely dominated by mapping retrieval attempts that fail to return existing physical to virtual address mappings. This is to be expected in workloads that have many tasks that do not share many pages such as the is-pomp workload. Prefetch is more useful in cases where there are virtual to physical mappings to retrieve, and those mappings will subsequently be used therefore removing faults from the workloads execution. While the

average time necessary to retrieve a mapping increases with greater amounts of prefetch, the overall results do not show significant degradation since the duration of the is-pomp process is many orders of magnitude greater than the total mapping fetch overhead.

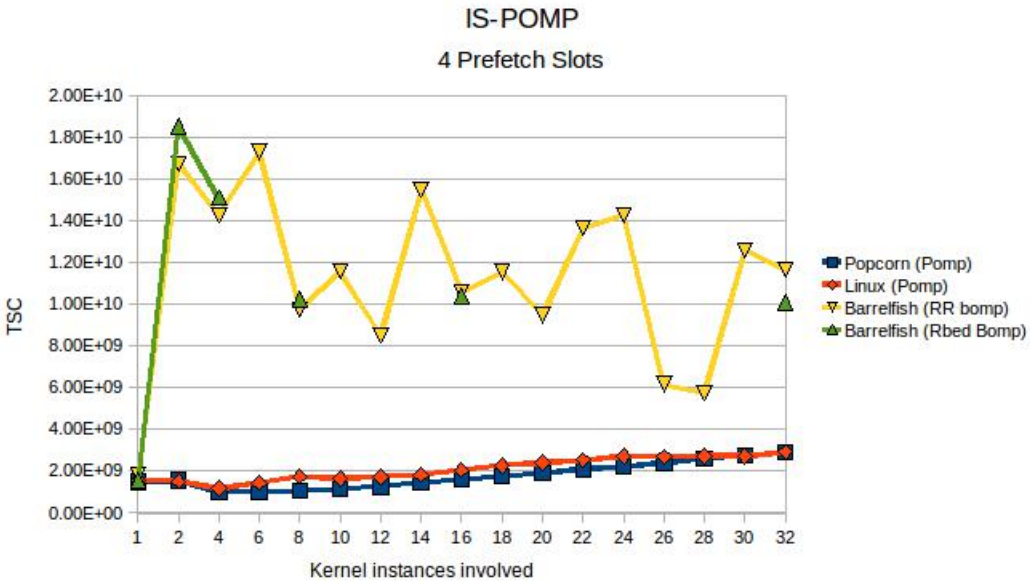

Figure 6.13: is-pomp results varying involved kernel instances with 4 prefetch slots

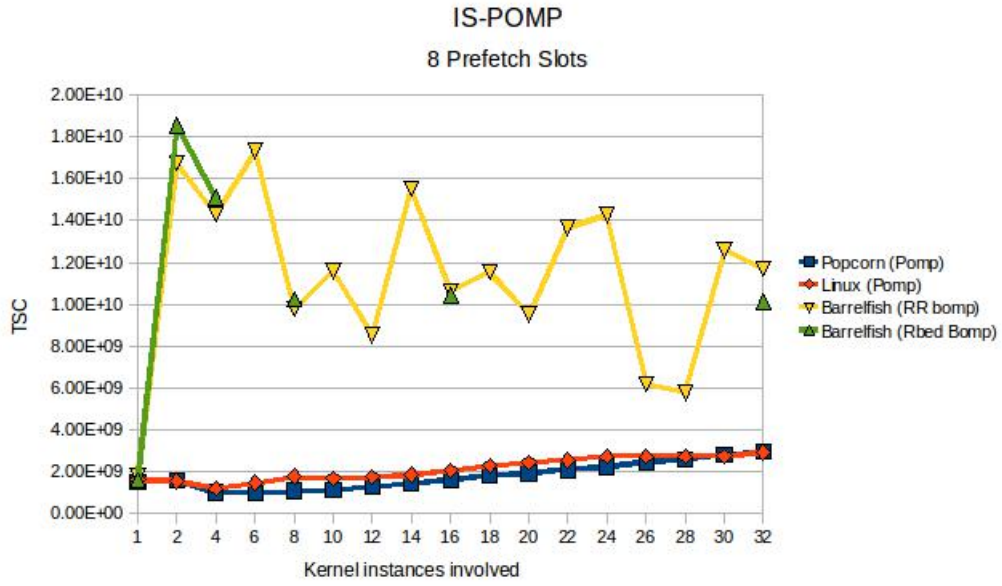

Figure 6.14: is-pomp results varying involved kernel instances with 8 prefetch slot

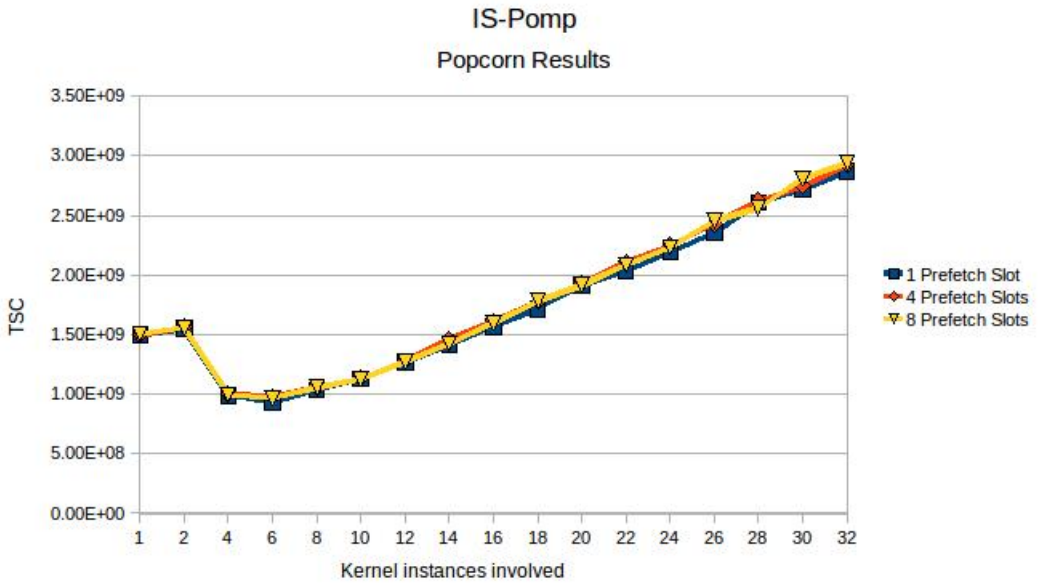

Figure 6.15: is-pomp results varying kernel instances involved

The Popcorn results indicate improved scaling over both SMP Linux and Barrelfish OS for the IS-POMP workload over the first 32 kernel instances. This indicates that the overhead that was removed from SMP Linux in the form of spin lock contention exceeds the overhead that was added to implement mapping replication and other Popcorn features.

6.3 FT-POMP

The ft-pomp workload is a parallel discrete 3D fast Fourier Transform benchmark designed to test all to all communications between computing nodes. Ft-pomp is a Popcorn adaptation of the ft-omp workload designed by NASA. The following section describes the ft-pomp workload in order to give the reader an understanding of what aspects of Popcorn this benchmark exercises most. Once the workload profile is explained, the ft-pomp Popcorn results are presented and are then explained. Popcorn performance is also contrasted with the performance of the same workload on SMP Linux.

6.3.1 FT-POMP Workload Profile

The ft-pomp workload can be run across as many kernel instances as are present. When the ft-pomp program is invoked, the number of kernel instances to use are specified as a command-line argument. The ft-pomp workload performs two task migrations for every

kernel instance that is used beyond the first kernel instance. If only one kernel instance is used, no task migrations occur, and all processing is done in the kernel instance on which the program was invoked. The number of migrations for an invocation of ft-pomp involving N kernel instances is therefore 2(N-1).

Figure 6.16 shows the frequency of various significant events as the workload is executed on a varying number of kernel instances. The event types were chosen to show overhead specific to features introduced to the Linux kernel by Popcorn for task migration and address space maintenance. At low numbers of involved kernels, the Popcorn overhead is dominated by mapping fetches that result in the retrieval of pages that are file-backed with no PTEs already mapped. This is the result of a large file-backed memory map operation that takes place in user space. The frequency of this type of event reduces slightly as more kernel instances are introduced. As the number of kernel instances involved increases, increasing amounts of Popcorn overhead is introduced involving retrieval of file-backed mappings that already have PTEs. At approximately 8 kernel instances, the number of retrievals of mappings with PTEs outnumber the number of mappings without PTEs.

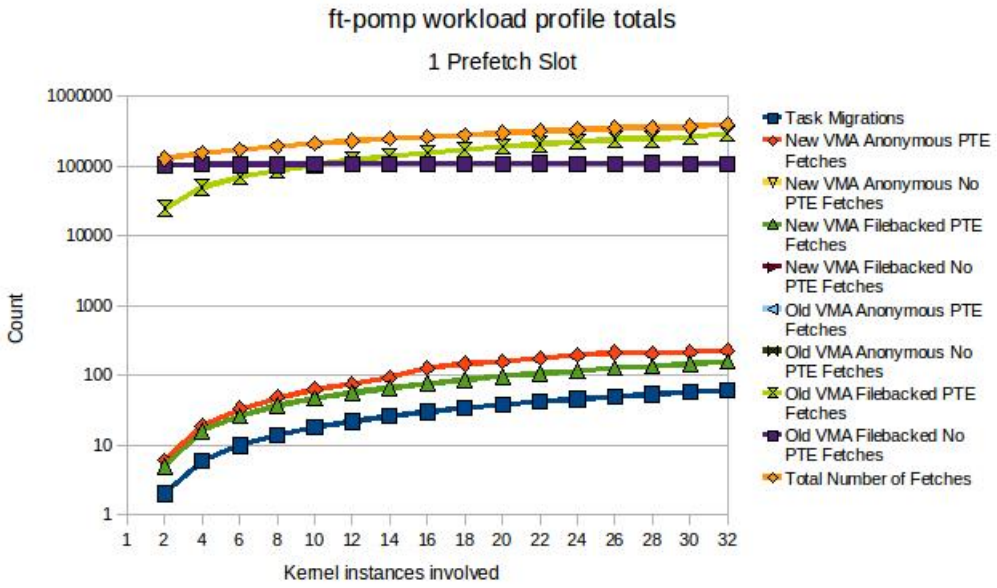

Figure 6.16: FT Workload Event Totals With 1 Prefetch Slot

Figure 6.17 shows details on how the totals shown above break down per kernel instance for the specific case where 16 kernel instances are involved. This graph shows that the vast majority of mapping retrievals for which no PTE was found are concentrated on the originating kernel instance, kernel instance 0. A small but constant number are found on all other kernel instances involved. It also shows that the number of file-backed mappings with PTEs are constant across all kernel instances except the originating kernel instance.

The constant number of retrievals of this type explains its increasing dominance in seen in Figure 6.16.

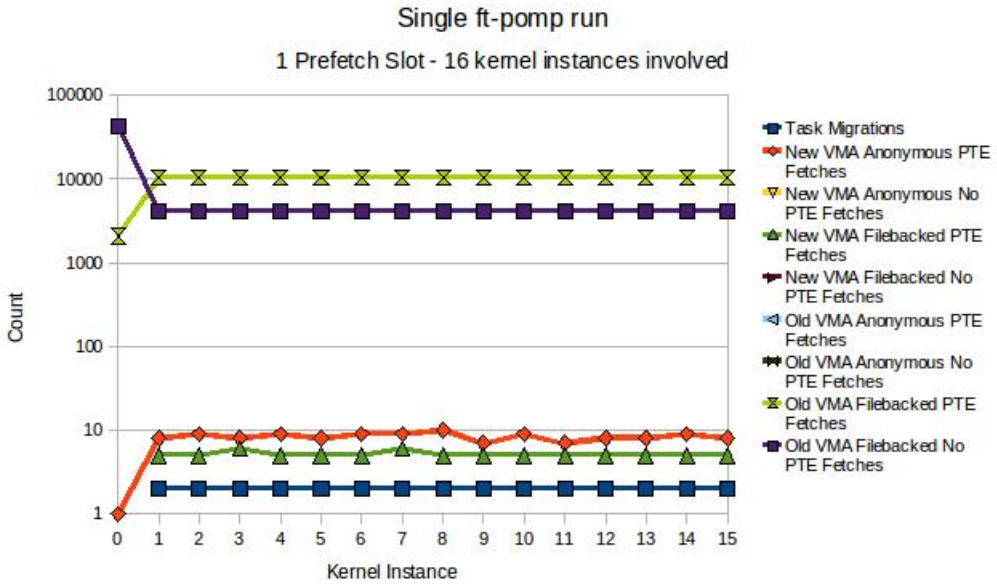

Figure 6.17: FT Workload profile with 16 kernel instances with 1 prefetch slot

Table 6.6: FT-POMP fault processing times by kernel instance with 1 prefetch slot

CPU	Count	Max	Min	Average
0	45362	37035267	61783	215503
1	14509	24067468	50823	211946
2	14510	26124144	63208	238298
3	14517	22030107	63357	234813
4	14512	21720829	66053	236779
5	14509	26096374	62088	230393
6	14511	21861133	64916	235760
7	14518	26133684	68246	240183
8	14509	26201378	69235	239949
9	14508	22387097	70389	341393
10	14512	23900163	70386	235812
11	14511	23913635	70194	244110
12	14513	23896782	66617	237022
13	14510	24002677	68510	233996
14	14513	24009707	69174	229707
15	14511	21902086	61259	224929

The following figures show how this workload responds to increasing the amount of prefetch that is performed during each mapping retrieval. 4 prefetch slot and 8 prefetch slot profiles are shown in these graphs, as opposed to the single prefetch slot that is shown above. Because the ft-pomp workload requires the retrieval of already existing file-backed mappings with mapped PTEs, this workload profile does change visibly. Comparing the 4 and 8 prefetch slot graphs with the 1 prefetch slot, it is clear that the number of fetches for mapping with PTEs decreases drastically with increased prefetch. This is consistent with the observation that many prefetch operations will retrieve mappings that will subsequently be used by the retrieving kernel instance. When this happens, it is saved the effort of faulting and performing another mapping retrieval operation.

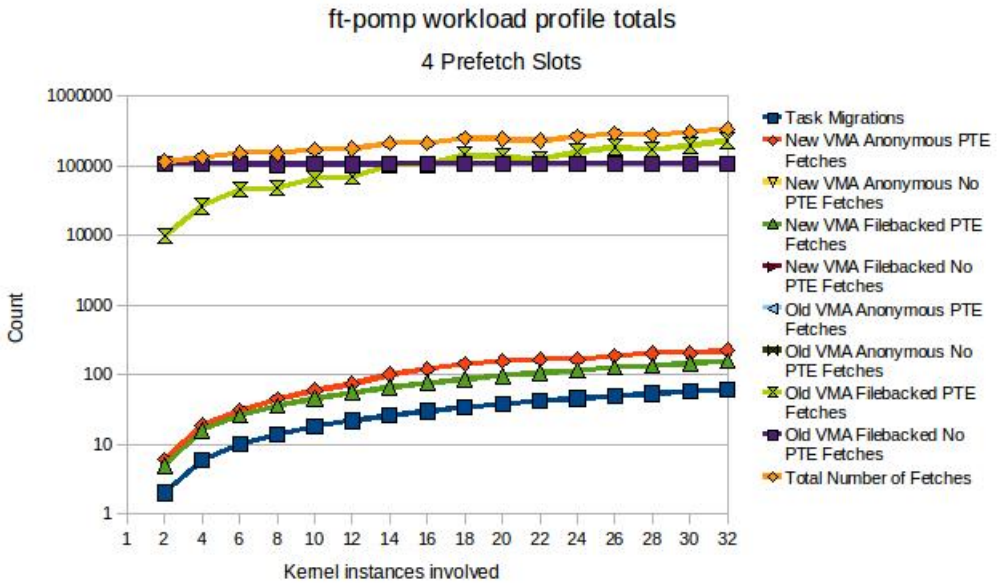

Figure 6.18: FT Workload Event Totals With 4 Prefetch Slots

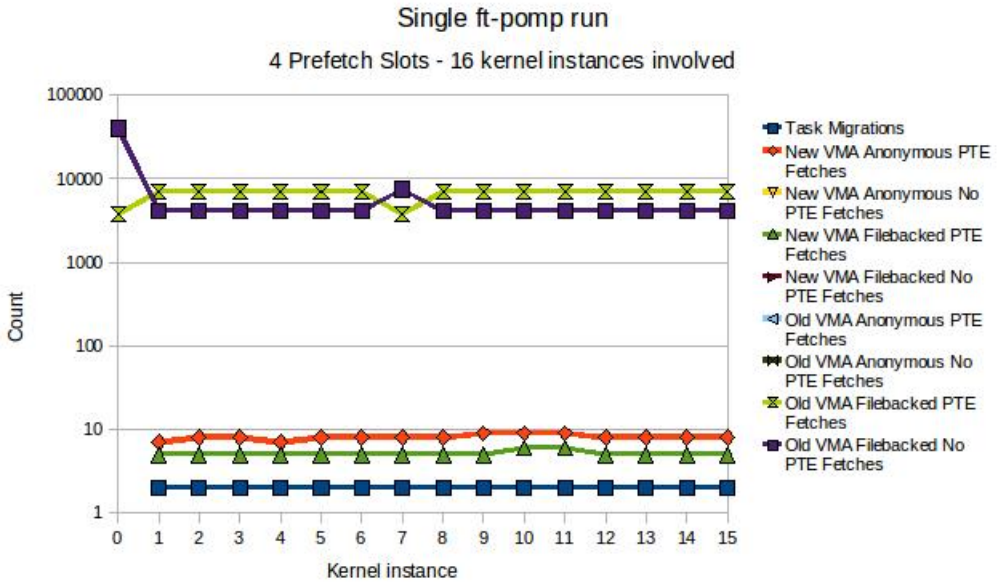

Figure 6.19: FT Workload profile with 16 kernel instances with 4 prefetch slots

As the amount of prefetch increases, the number of fetches decreases. This is because many of the mapping retrievals that are done result in those with PTEs. So the prefetched PTEs reduces the quantity of faults that occur. However, while the number of fetches decreases,

the amount of time required to perform the fetch increases.

Table 6.7: FT-POMP fault processing times by kernel instance with 4 prefetch slots

CPU	Count	Max	Min	Average
0	44146	242362499	62593	312592
1	11176	119324934	59473	487471
2	11146	121340628	62505	449252
3	11145	156467668	62101	494811
4	11178	243830341	70759	629325
5	11174	236680246	71869	666358
6	11176	120573366	58972	481549
7	11302	277380376	75708	1147431
8	11176	117630478	68216	512174
9	11148	52094917	58356	465855
10	11180	165076351	66035	522691
11	11151	157490307	65333	490592
12	11149	156527804	63694	468820
13	11148	71723982	70327	457169
14	11145	41434740	67006	445696
15	11143	40331105	70866	445610

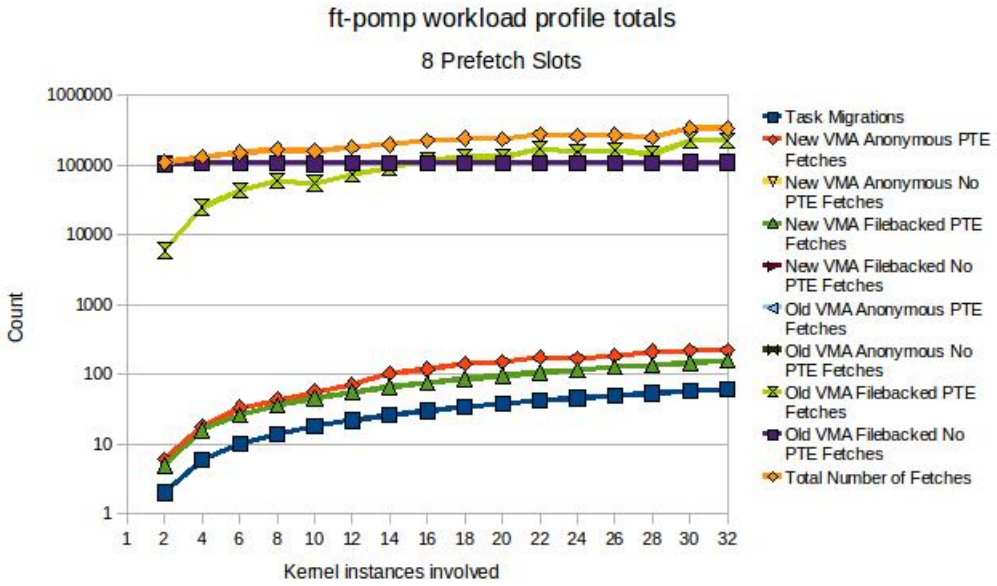

Figure 6.20: FT Workload Event Totals With 8 Prefetch Slots

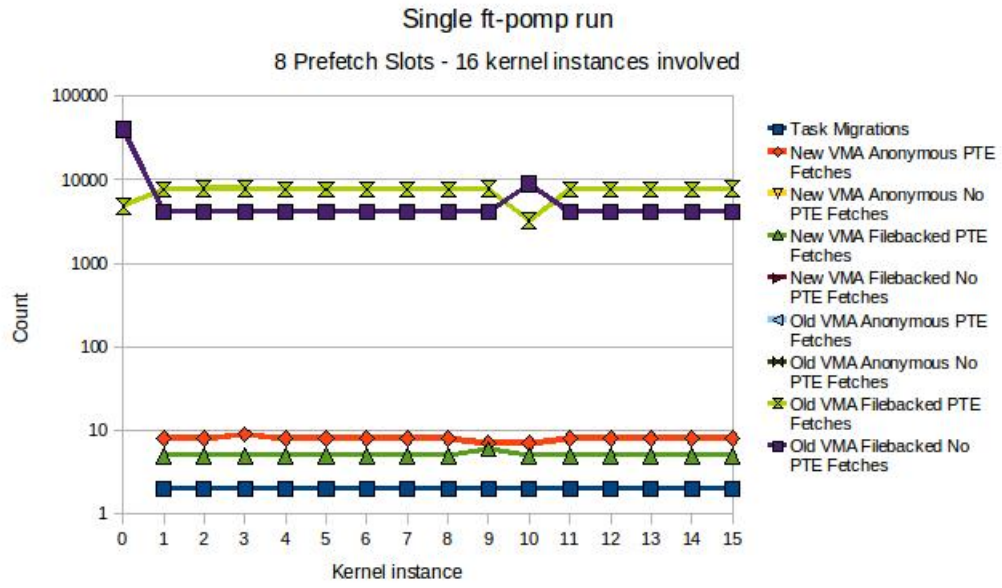

Figure 6.21: FT Workload profile with 16 kernel instances with 8 prefetch slots

Table 6.8: FT-POMP fault processing times by kernel instance with 8 prefetch slots

CPU	Count	Max	Min	Average
0	45297	241161885	62222	370667
1	11932	592186802	61734	544270
2	12035	156764226	67403	514542
3	12028	242229984	73296	597751
4	11905	152836356	71318	490511
5	11897	107426744	62665	482112
6	11900	104845746	66263	502926
7	11897	202402926	64300	556359
8	11900	159180796	61150	508056
9	12034	362408620	75870	1031641
10	12154	283801594	69753	1156403
11	11902	67254063	71068	477047
12	11902	117274695	75021	515296
13	11904	88057942	66813	494977
14	11897	76603845	77661	485151
15	12025	321979307	74999	628735

For the average ft-pomp execution involving 2 kernels, the messaging overhead is 45%, 30%,

and 31% for 1 prefetch slot, 4 prefetch slots, and 8 prefetch slots respectively. This percentage decreases due to the increased overhead associated with looking up mappings, and then installing them at their destination.

Table 6.9: FT-POMP 2 Mapping Retrieval Message Transport Times

	1 Prefetch Slot	4 Prefetch Slots	8 Prefetch Slots
Avg Mapping Fetch Processing (TSC)	40690	73068	64785
Total Message Transit (TSC)	18529	22056	20185
Amount Of Mapping Fetch Processing Waiting For Msg Transit (%)	45.53	30.18	31.15

The perf tool was used to understand which overhead components were most significant in both SMP Linux and Popcorn, and see which ones increased and decreased between the two platforms. These measurements were taken on the main thread of the workload. The greatest kernel-mode overhead contributors are shown graphed below, as well as the remaining kernel-mode overhead, and the breakdown of time spent in user-mode versus kernel-mode.

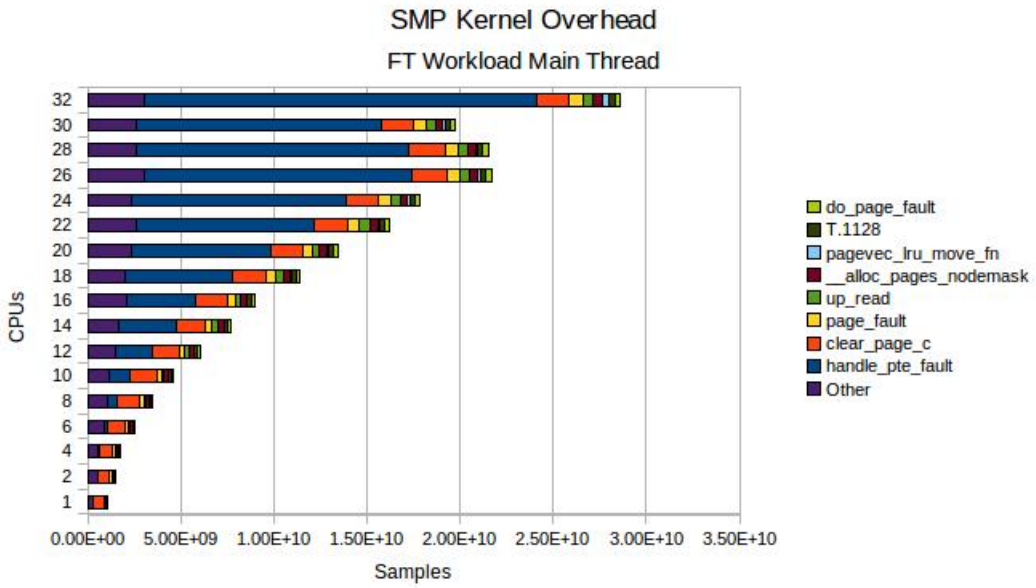

Figure 6.22: Perf Measurement for FT-POMP Workload on SMP Linux

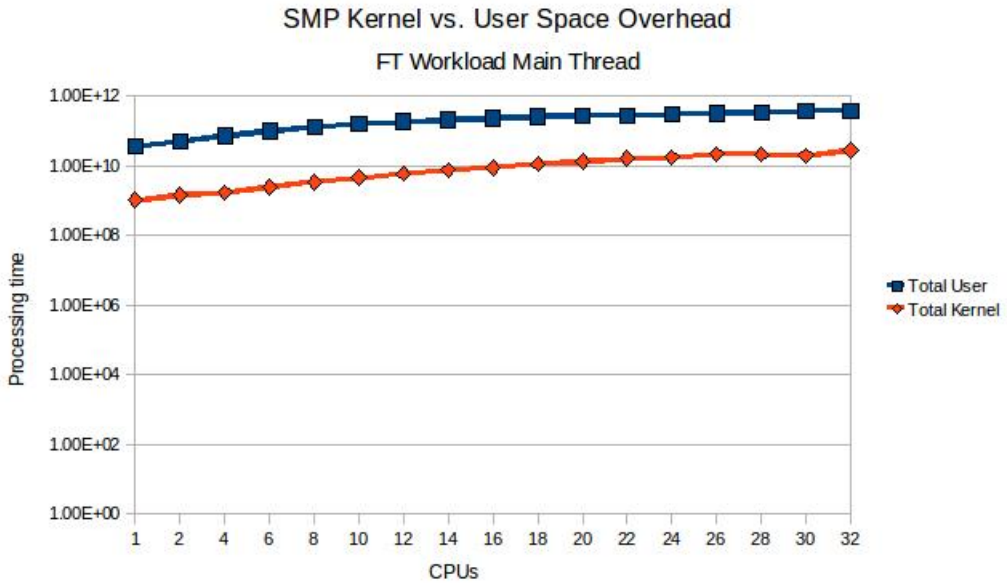

Figure 6.23: Perf Measurement for FT-POMP Workload on SMP Linux - User vs Kernel

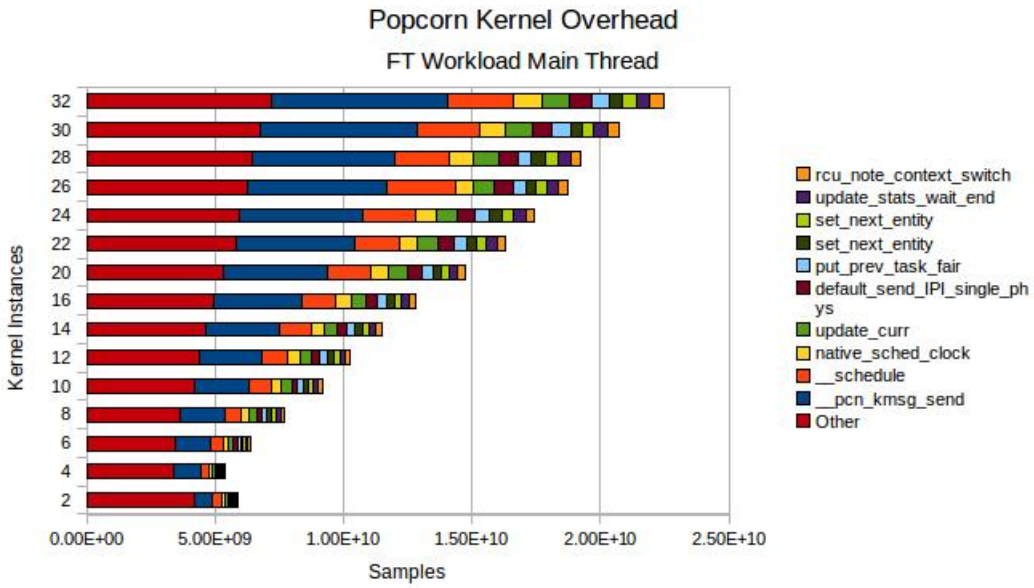

Figure 6.24: Perf Measurement for FT-POMP Workload on Popcorn

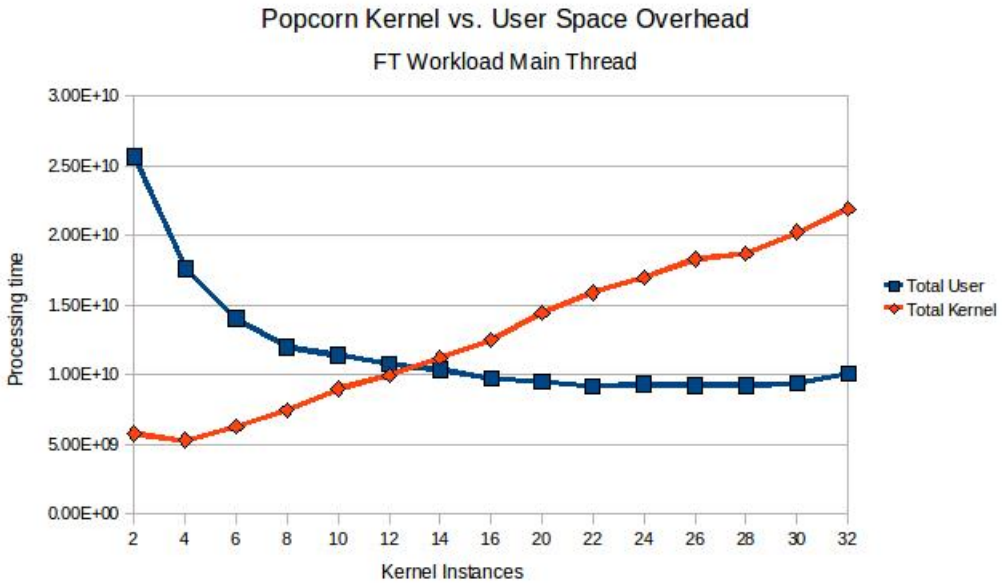

Figure 6.25: Perf Measurement for FT-POMP Workload on Popcorn - User vs Kernel

SMP Kernel overhead is increasingly dominated by time spent in *handle_pte_fault*. This function is called during the page mapping process. That function creates significant overhead due to contention for a spin lock. As the number of cores contending for that spin lock increase, contention also increases. Like IS, the other source of contention is the *clear_page_c* function. These functions are still sources of overhead in Popcorn, but they fall out of the list of most significant overhead contributors, being replaced by *__pcn_kmsg_send*, and *__schedule*. The send function, is part of Popcorn's transport layer. Schedule becomes significant due to the fact that while a kernel instance is waiting for a response from a remote kernel instance, it waits by scheduling. If there are other tasks to process during that time, they will execute. If not, control will be returned back to the waiting task where another scheduler invocation may occur. In Popcorn, so much time is spent waiting on responses and sending requests, that eventually that time overtakes the time spent in user space on the main thread.

6.3.2 FT-POMP Results

Of all the workloads, ft-pomp has both the highest fault retrieval requirements, and the longest duration in time. Popcorn Linux is able to keep up with SMP Linux. When prefetch is increased from 1 slot to 4 and 8 slots, Popcorn performance surpasses SMP Linux between the times where 10 and 18 kernel instances are involved in the workload processing. Before that time and after, SMP Linux outperforms Popcorn Linux.

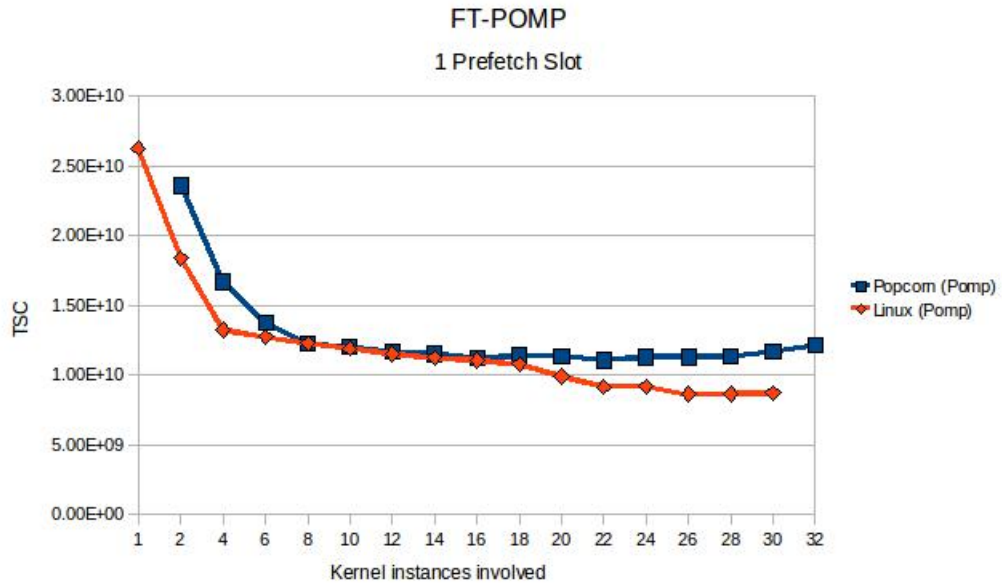

Figure 6.26: ft-pomp results varying involved kernel instances with 1 prefetch slot

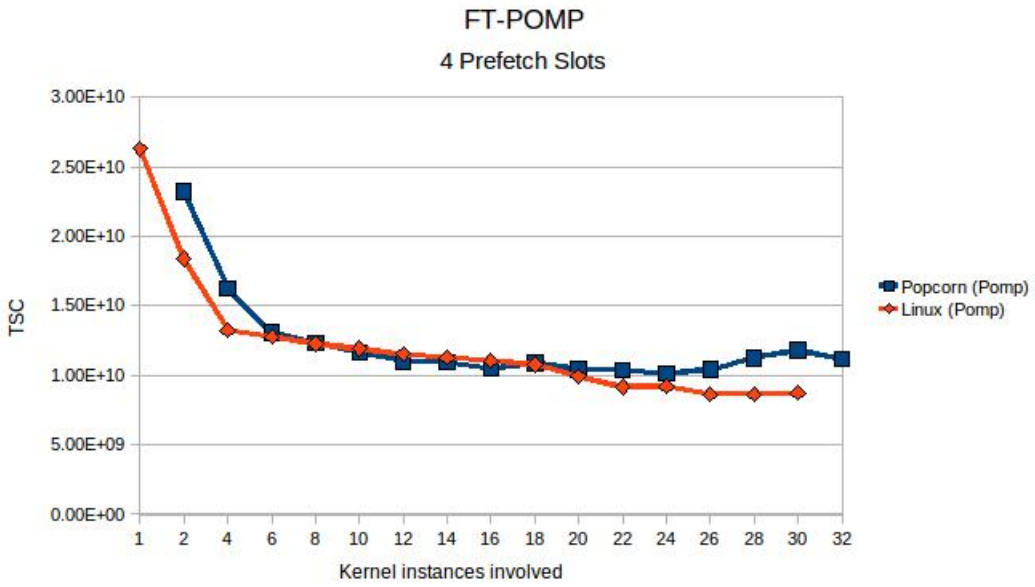

Figure 6.27: ft-pomp results varying involved kernel instances with 4 prefetch slots

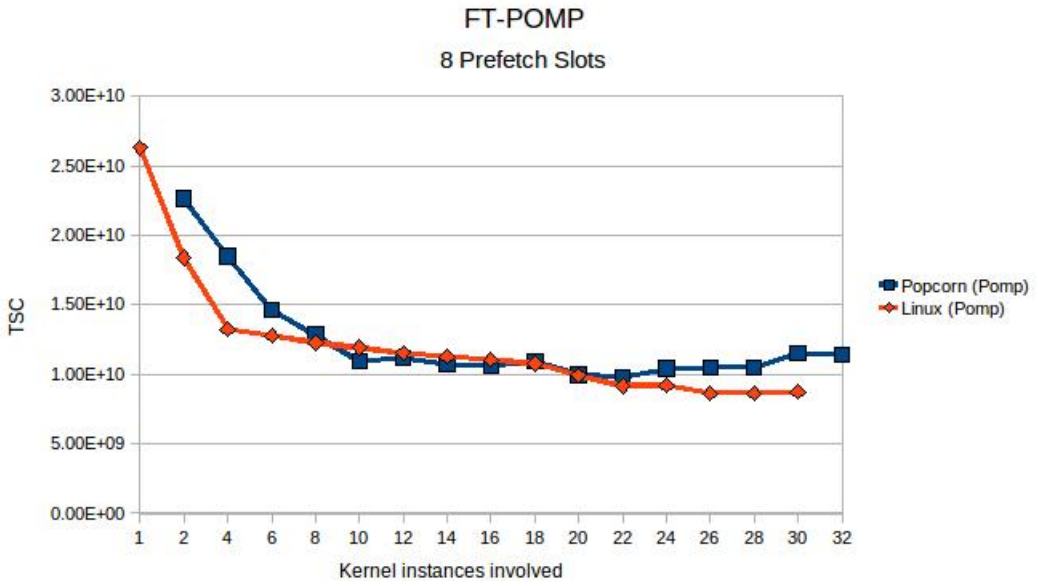

Figure 6.28: ft-pomp results varying involved kernel instances with 8 prefetch slot

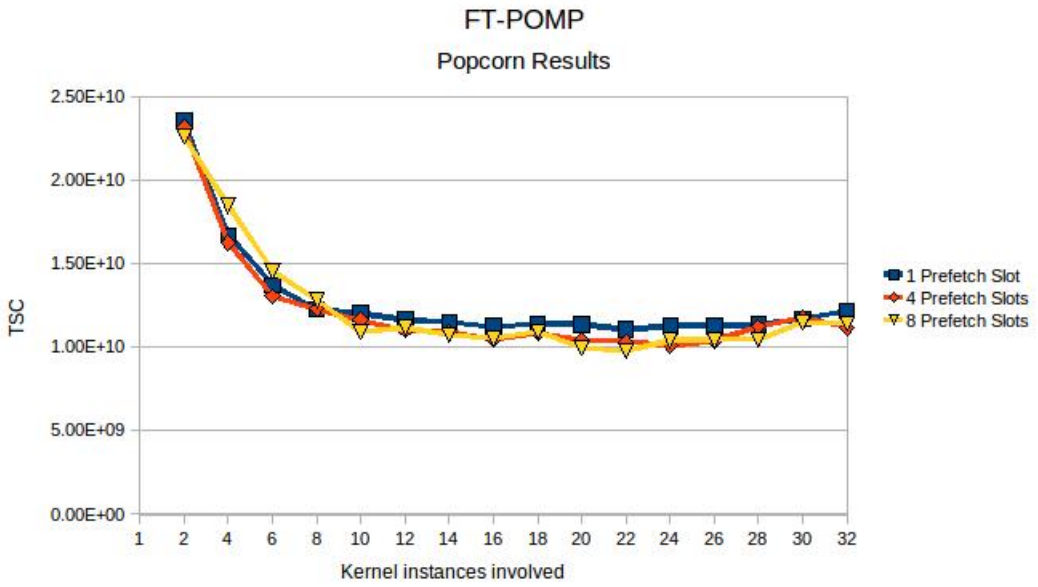

Figure 6.29: ft-pomp results varying kernel instances involved

SMP Linux scales better than Popcorn as the number of kernel instances increase to 32 for the FT-POMP workload, except a period between 10 kernel instances and 18 kernel instances when higher prefetch is used. This indicates that Popcorn's high messaging and message

processing costs ware too significant to be overcome by the removal of the SMP memory subsystem spin lock contention such as the one found in *handle_pte_fault*.

6.4 CG-POMP

The cg-pomp workload is a parallel conjugate gradient benchmark designed to test irregular memory access. Cg-pomp is a Popcorn adaptation of the cg-omp workload designed by NASA. The following section describes the cg-pomp workload in order to give the reader an understanding of what aspects of Popcorn this benchmark exercises most. Once the workload profile is explained, the cg-pomp Popcorn results are presented and are then explained. Popcorn performance is also contrasted with the performance of the same workload on SMP Linux, and Barrelfish OS.

6.4.1 CG-POMP Workload Profile

The cg-pomp workload can be run across as many kernel instances as are present. When the cg-pomp program is invoked, the number of kernel instances to use are specified as a command-line argument. The cg-pomp workload performs five task migrations for every kernel instance that is used beyond the first kernel instance. If only one kernel instance is used, no task migrations occur, and all processing is done in the kernel instance on which the program was invoked. The number of migrations for an invocation of cg-pomp involving N kernel instances is therefore 5(N-1).

Figure 6.30 shows the frequency of various significant events as the workload is executed on a varying number of kernel instances. The event types were chosen to show overhead specific to features introduced to the Linux kernel by Popcorn for task migration and address space maintenance. The Popcorn overhead is again dominated by the retrieval of file-backed mappings without assigned PTEs. Retrievals of file-backed mappings with PTEs are fewer, though still sizable, and become more predominant as the number of kernels involved increase.

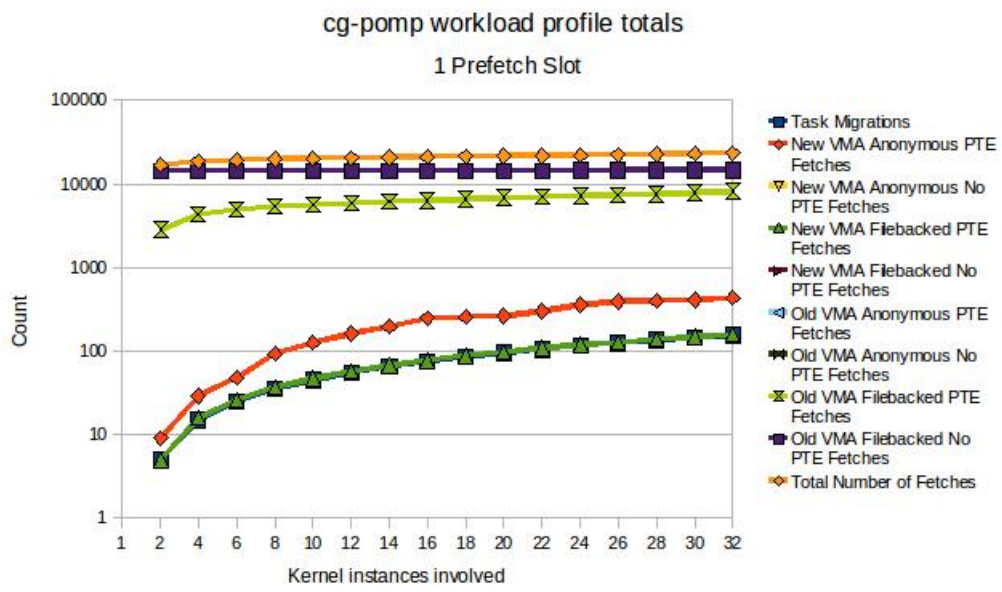

Figure 6.30: CG Workload Event Totals With 1 Prefetch Slot

Figure 6.31 shows details on how the totals shown above break down per kernel instance for the specific case where 16 kernel instances are involved. This graph shows that all file-backed mapping retrievals resulting in no PTEs are concentrated on same kernel instance, where the program was invoked - kernel instance 0. The mapping retrievals that result in PTEs are few and spread evenly across the remaining kernel instances.

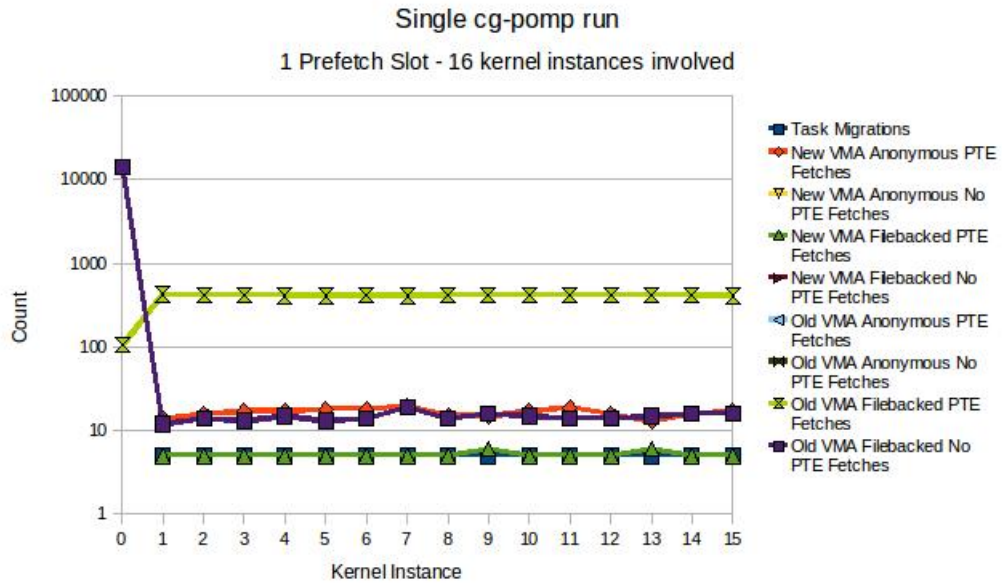

Figure 6.31: CG Workload profile with 16 kernel instances with 1 prefetch slot

Table 6.10: CG-POMP fault processing times by kernel instance with 1 prefetch slot

CPU	Count	Max	Min	Average
0	14198	20693485	61217	72295
1	455	15709167	64430	622207
2	452	22071285	79570	943404
3	454	21825769	67607	885948
4	448	21706031	77262	795119
5	444	21838690	76625	861511
6	453	21917321	72825	773319
7	449	35763869	80860	1068730
8	453	21831829	67679	720923
9	455	21914191	81722	936710
10	451	22342296	77688	761581
11	456	21792512	66393	806685
12	453	21760444	71150	634628
13	451	21906196	62413	786522
14	450	21793997	79600	681191
15	445	21850220	76877	730262

The following figures show how this workload responds to increasing the amount of prefetch

that is performed during each mapping retrieval. 4 prefetch slot and 8 prefetch slot profiles are shown in these graphs, as opposed to the single prefetch slot that is shown above. As the amount of prefetch is increased, the number of faults that result in the migration of existing PTEs decreases, as expected. Also as expected, the number of faults resulting in responses indicating no PTE is not affected as prefetch increases.

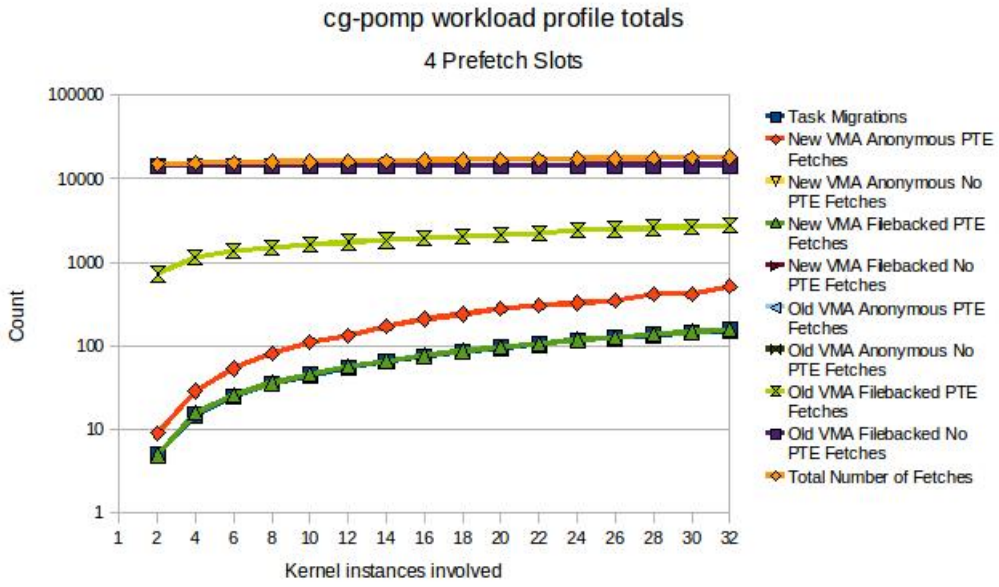

Figure 6.32: CG Workload Event Totals With 4 Prefetch Slots

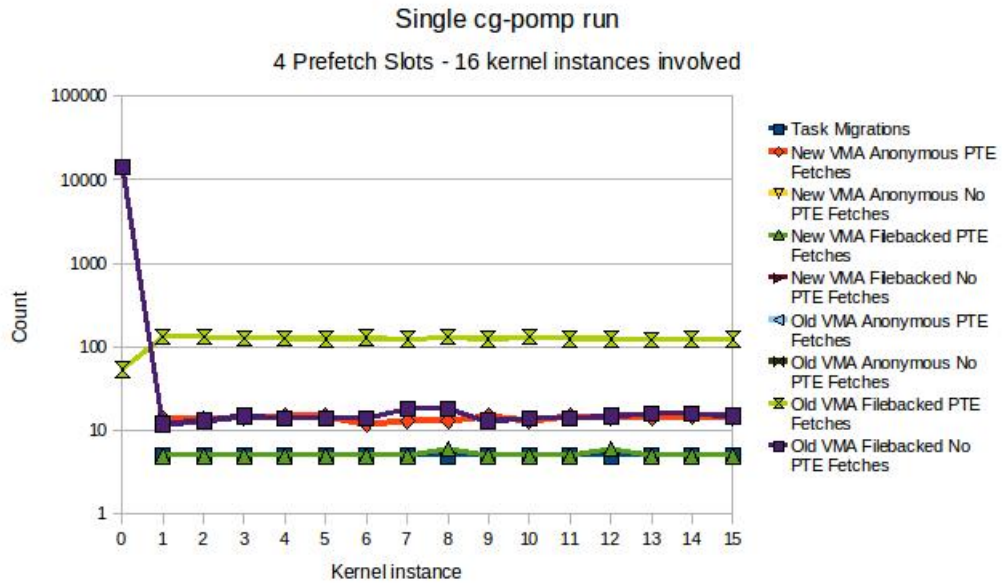

Figure 6.33: CG Workload profile with 16 kernel instances with 4 prefetch slots

As seen in previous workloads, as the number of fetches decreases, the amount of time required to perform the fetch increases.

Table 6.11: CG-POMP fault processing times by kernel instance with 4 prefetch slots

CPU	Count	Max	Min	Average
0	14157	11909728	61698	75076
1	164	22419820	76808	802292
2	164	31230796	83175	1009647
3	161	22693753	98780	928712
4	162	22317714	87423	964857
5	158	16022599	90936	682330
6	160	22363001	88248	1012737
7	159	22334320	101406	1151013
8	170	20266322	75444	829762
9	157	22318944	89587	1355735
10	166	23170570	81430	1393219
11	159	23128337	76181	1095501
12	159	20115246	64675	1243417
13	155	28435287	100816	1387345
14	159	12583480	104096	759596
15	159	28354020	78326	1166056

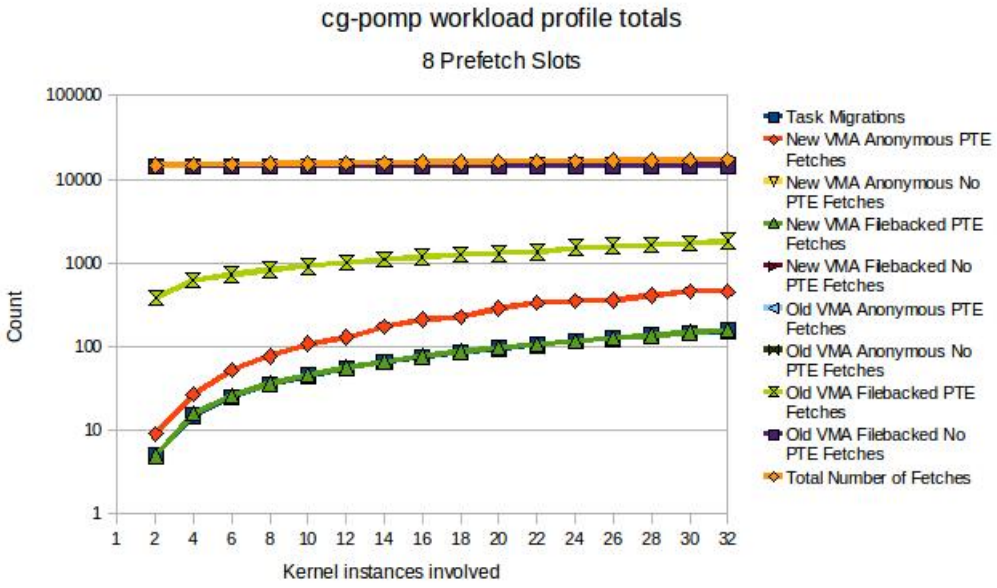

Figure 6.34: CG Workload Event Totals With 8 Prefetch Slots

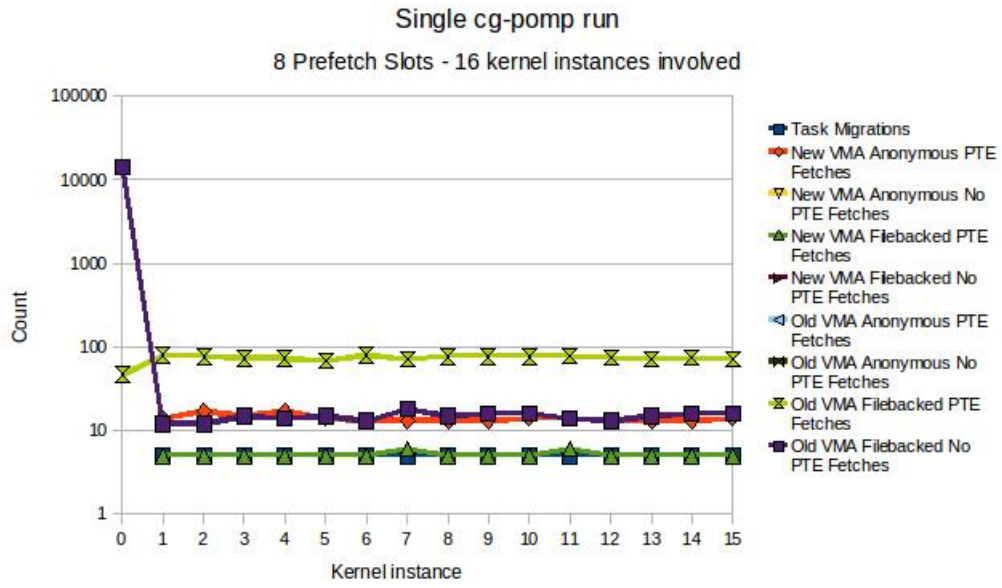

Figure 6.35: CG Workload profile with 16 kernel instances with 8 prefetch slots

Table 6.12: CG-POMP fault processing times by kernel instance with 8 prefetch slots

CPU	Count	Max	Min	Average
0	14151	13470976	61442	81150
1	110	18166532	87343	963320
2	111	21722208	101956	1198140
3	108	21864295	109687	1360902
4	109	21853583	100993	1497228
5	103	21738681	94278	1170927
6	111	21893953	95621	1747774
7	109	33172637	91359	2035906
8	110	24667779	91913	1734664
9	111	23223111	103707	1951710
10	112	22609703	75427	1605889
11	112	22171975	126986	1908350
12	105	21904695	106484	1564267
13	104	25131773	113929	2348502
14	108	22169115	90320	1641559
15	107	21847412	96262	1790908

For the average cg-pomp execution involving 2 kernels, the messaging overhead is 59%,

47%, and 40% for 1 prefetch slot, 4 prefetch slots, and 8 prefetch slots respectively. This percentage decreases due to the increased overhead associated with looking up mappings, and then installing them at their destination.

Table 6.13: CG-POMP 2 Mapping Retrieval Message Transport Times

	1 Prefetch Slot	4 Prefetch Slots	8 Prefetch Slots
Avg Mapping Fetch Processing (TSC)	34269	46800	56051
Total Message Transit (TSC)	20477	22299	22673
Amount Of Mapping Fetch Processing Waiting For Msg Transit (%)	59.75	47.64	40.45

The perf tool was used to understand which overhead components were most significant in both SMP Linux and Popcorn, and see which ones increased and decreased between the two platforms. These measurements were taken on the main thread of the workload. The greatest kernel-mode overhead contributors are shown graphed below, as well as the remaining kernel-mode overhead, and the breakdown of time spent in user-mode versus kernel-mode.

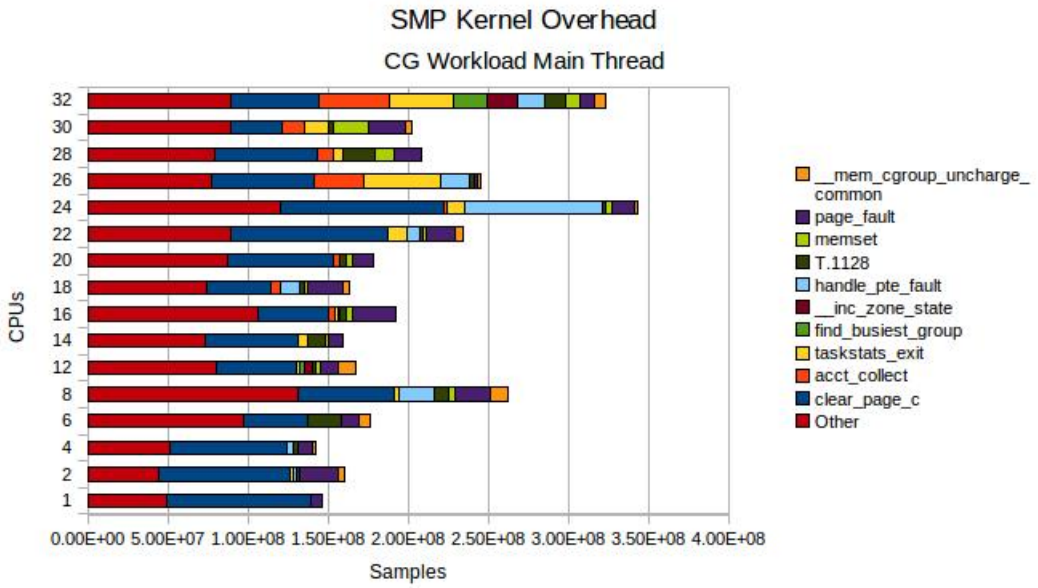

Figure 6.36: Perf Measurement for CG-POMP Workload on SMP Linux

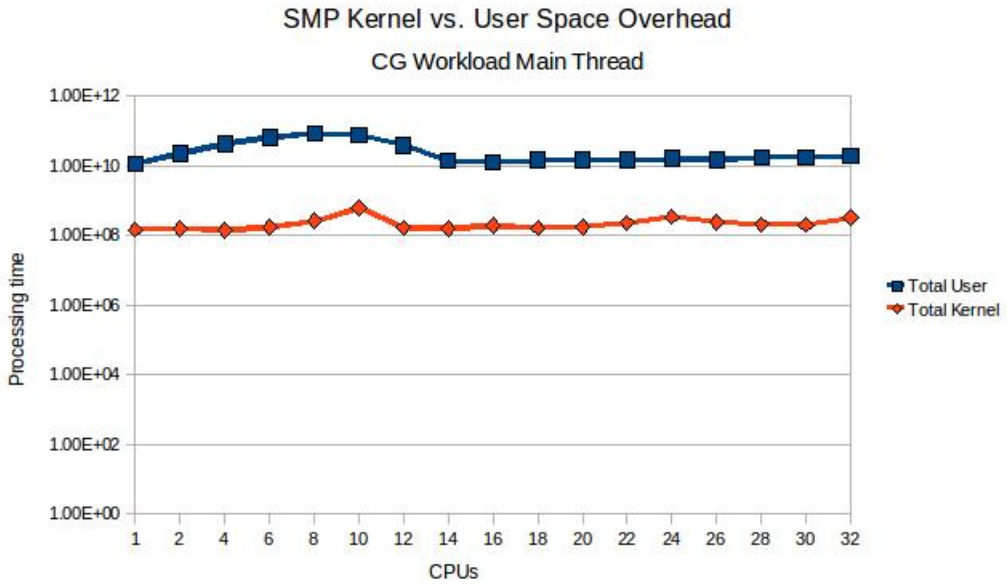

Figure 6.37: Perf Measurement for CG-POMP Workload on SMP Linux - User vs Kernel

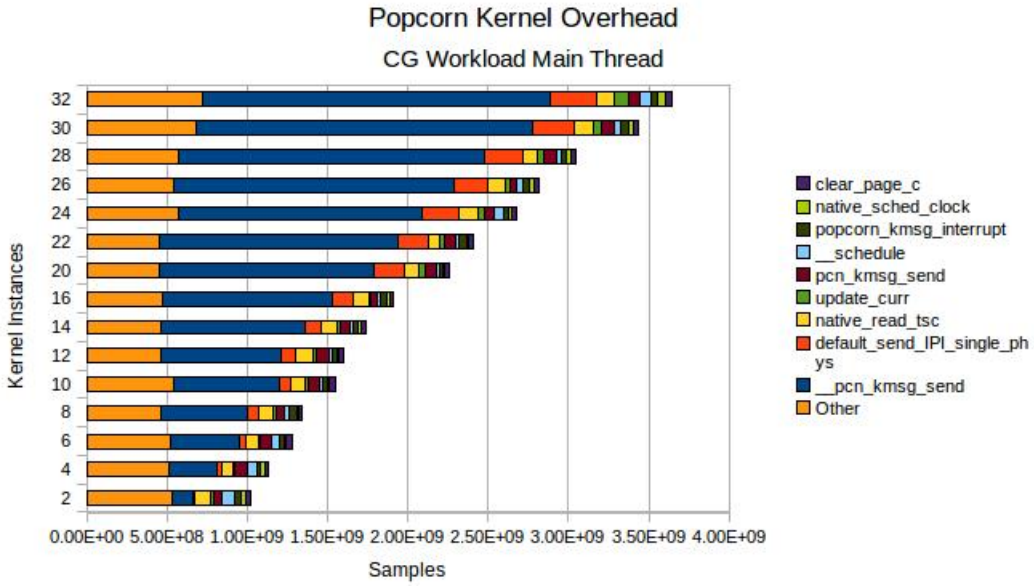

Figure 6.38: Perf Measurement for CG-POMP Workload on Popcorn

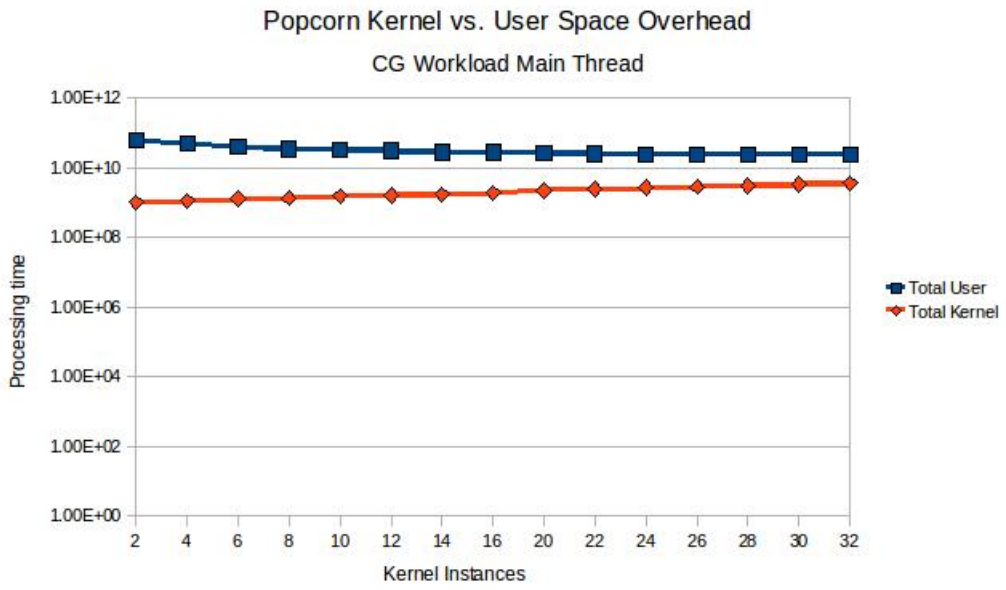

Figure 6.39: Perf Measurement for CG-POMP Workload on Popcorn - User vs Kernel

SMP Kernel overhead is very modest when running the CG workload, being dominated by *clear_page_c*. The Popcorn overhead is an order of magnitude higher, and like the previous Popcorn workloads is dominated by the transport. Both *__pcn_kmsg _send* and *default_send_IPI_single_phys* are parts of the transport.

6.4.2 CG-POMP Results

Popcorn significantly under-performs both SMP Linux and Barrelfish OS when processing the cg-pomp workload. While the Popcorn curve follows the SMP Linux and Barrelfish OS curves, it has a positive offset and never catches up to the other operating systems. When two kernel instances are involved, a large performance hit is taken as kernel instance 0 works to allocate PTEs to its address space, since it so rarely receives any already mapped from other kernel instances. As the number of kernel instances increases, the quantity of mapping requests that result in no migrated PTEs made by kernel instance 0 stays constant, and so the performance improves. The more kernel instances are introduced, the more compensation there is for the large performance overhead that kernel instance 0 experiences, and the better the performance becomes.

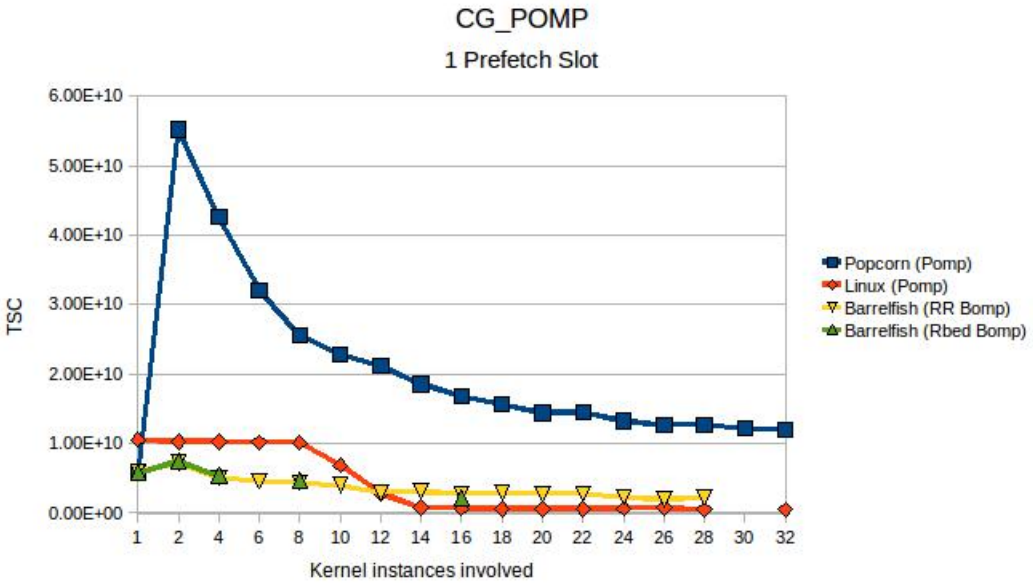

Figure 6.40: cg-pomp results varying involved kernel instances with 1 prefetch slot

As more prefetch is introduced, no significant performance gain is made due to the fact that the overhead is dominated by mapping retrievals that do not return a PTE, rather than those that do.

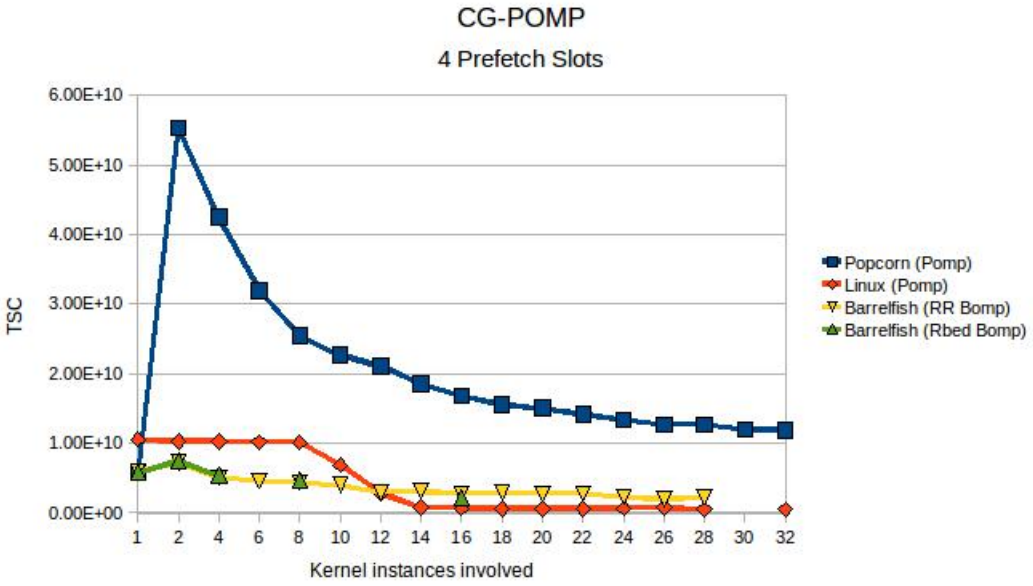

Figure 6.41: cg-pomp results varying involved kernel instances with 4 prefetch slots

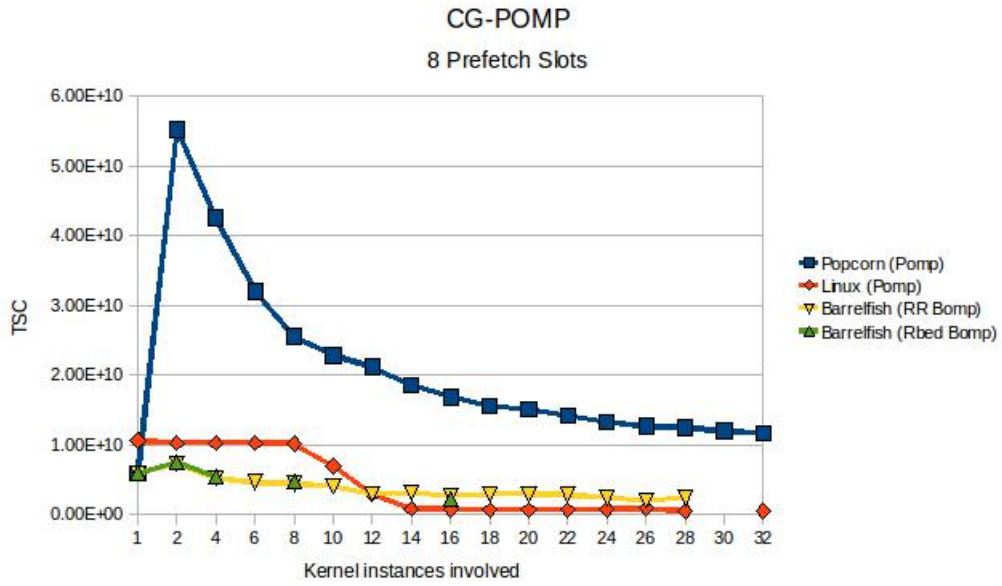

Figure 6.42: cg-pomp results varying involved kernel instances with 8 prefetch slot

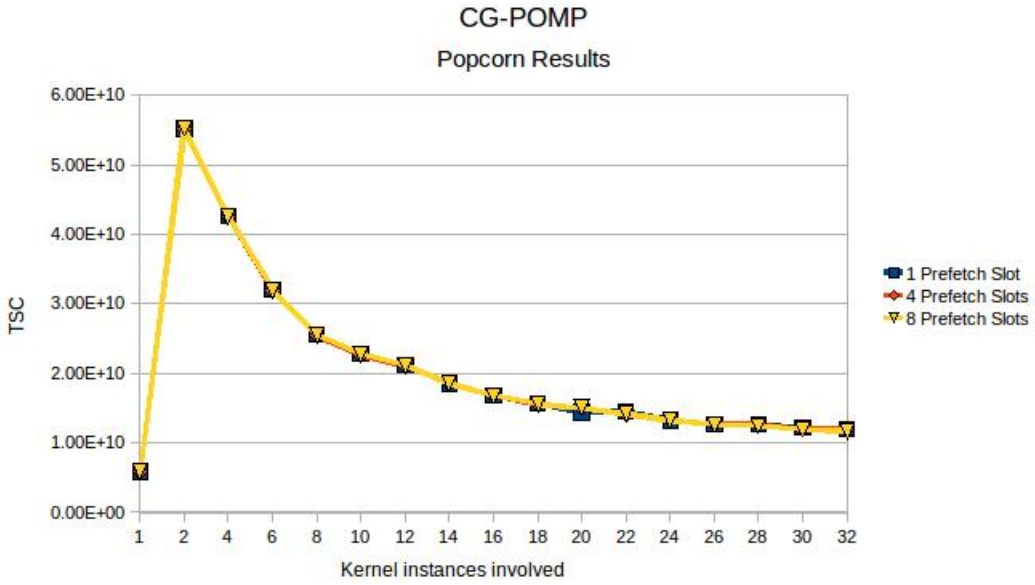

Figure 6.43: cg-pomp results varying kernel instances involved

SMP Linux scales significantly better than Popcorn as the number of kernel instances increases to 32 for the CG-POMP workload. The messaging and message processing overhead in Popcorn, when added to the other kernel overhead, exceeds that seen by the SMP Linux

kernel by a significant amount. This is driven by the high number of mapping requests made by kernel instance 0 that go unfulfilled by other kernel instances. The overhead of carrying out mapping requests is not balanced in this case by any gain associated with prefetch, as the number of remote mappings that can be used by kernel instance 0 is low. This is due to a mapping that is created on kernel instance 0, and used there before any remote kernel instances attempt to use them. When no remote kernel instances have any useful mapping information, the process of asking for mappings is wasted overhead.

6.5 BFS-POMP

The bfs-pomp workload is a parallel breadth-first search benchmark which is a graph traversal algorithm. Bfs-pomp is a Popcorn adaptation of the bfs benchmark, which is part of the Rodinia benchmark suite created by UVA[9]. The following section describes the bfs-pomp workload in order to give the reader an understanding of what aspects of Popcorn this benchmark exercises most. Once the workload profile is explained, the bfs-pomp Popcorn results are presented and are then explained. Popcorn performance is also contrasted with the performance of the same workload on SMP Linux.

6.5.1 BFS-POMP Workload Profile

The bfs-pomp workload can be run across as many kernel instances as are present. When the bfs-pomp program is invoked, the number of kernel instances to use are specified as a command-line argument. The bfs-pomp workload performs twelve task migrations for every kernel instance that is used beyond the first kernel instance. If only one kernel instance is used, no task migrations occur, and all processing is done in the kernel instance on which the program was invoked. The number of migrations for an invocation of bfs-pomp involving N kernel instances is therefore 12(N-1).

Figure 6.44 shows the frequency of various significant events as the workload is executed on a varying number of kernel instances. The event types were chosen to show overhead specific to features introduced to the Linux kernel by Popcorn for task migration and address space maintenance. Unlike the previously discussed workloads, the bfs-pomp workload is dominated by mapping requests which result in PTE migrations. As the number of kernel instances involved increases, the number of PTE migrations increase linearly.

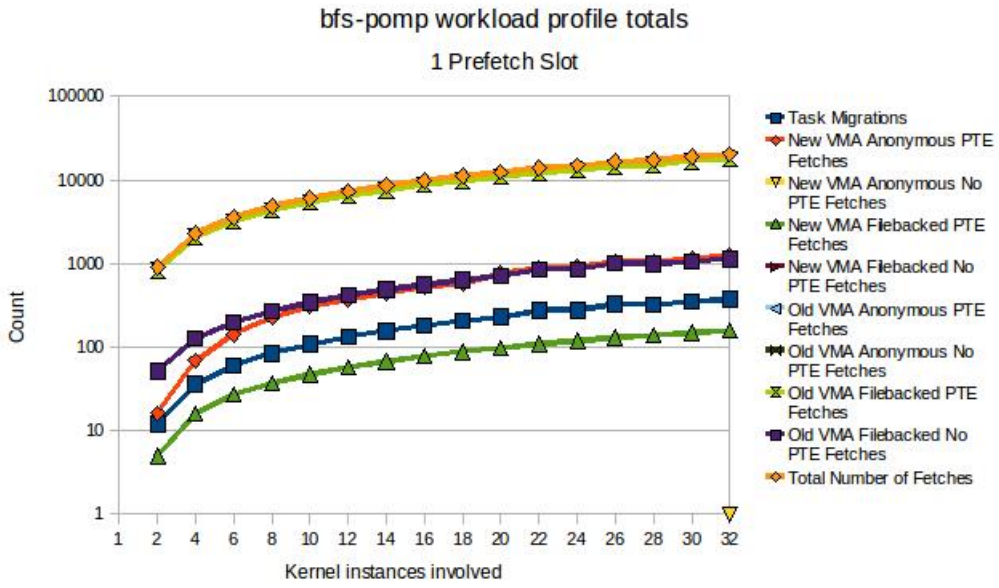

Figure 6.44: BFS Workload Event Totals With 1 Prefetch Slot

Figure 6.45 shows details on how the totals shown above break down per kernel instance for the specific case where 16 kernel instances are involved. This graph shows that the kernel instance 0 carries out many mapping retrievals that result in no PTEs. All other kernel instances carry out mapping retrievals that result in PTEs. The mapping retrievals that result in PTEs are evenly distributed among all kernel instances except kernel instance 0.

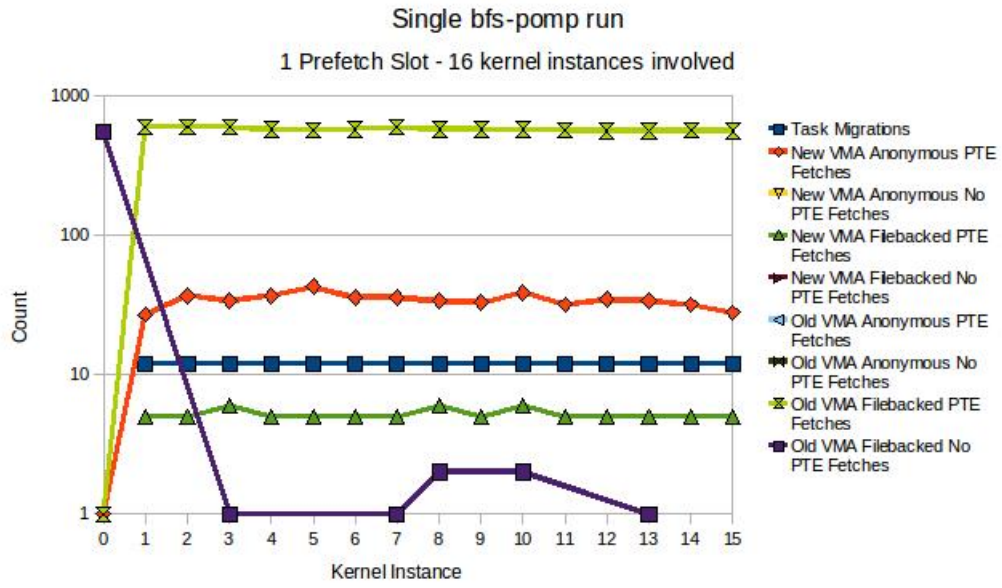

Figure 6.45: BFS Workload profile with 16 kernel instances with 1 prefetch slot

Table 6.14: BFS-POMP fault processing times by kernel instance with 1 prefetch slot

CPU	Count	Max	Min	Average
0	555	1430116	62174	72500
1	637	15315540	67887	285273
2	645	22501263	67904	303022
3	644	21757580	73404	407314
4	622	21798227	74452	360564
5	620	21762228	67071	311614
6	622	22084614	66160	369392
7	646	21779514	72548	297099
8	618	21813152	74138	358634
9	614	21744817	79233	321122
10	621	21952181	75100	331833
11	610	21787420	74215	348708
12	605	21709240	73445	337438
13	605	21898420	65573	296301
14	604	21681258	63776	317591
15	594	21737072	73808	289928

The following figures show how this workload responds to increasing the amount of prefetch

that is performed during each mapping retrieval. 4 prefetch slot and 8 prefetch slot profiles are shown in these graphs, as opposed to the single prefetch slot that is shown above. As the amount of prefetch is increased, the number of faults that result in the migration of existing PTEs decreases, as expected.

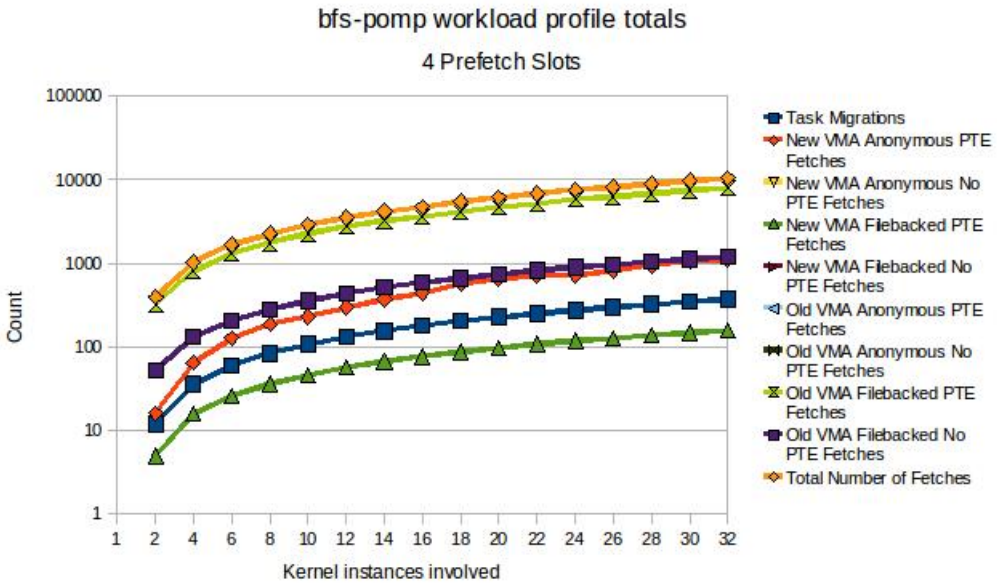

Figure 6.46: BFS Workload Event Totals With 4 Prefetch Slots

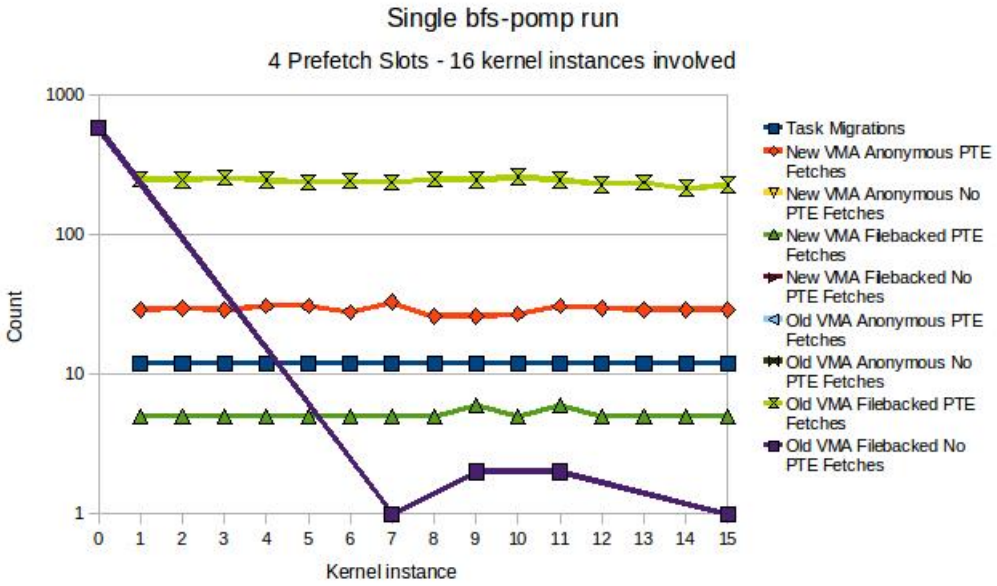

Figure 6.47: BFS Workload profile with 16 kernel instances with 4 prefetch slots

As seen in previous workloads, as the number of fetches decreases, the amount of time required to perform the fetch increases.

Table 6.15: BFS-POMP fault processing times by kernel instance with 4 prefetch slots

CPU	Count	Max	Min	Average
0	585	1358963	61149	69216
1	286	3225015	75266	282655
2	283	21772173	59794	507529
3	290	21777983	59773	524717
4	283	21881838	72838	531752
5	275	21780315	72137	404212
6	277	21849965	65423	449015
7	276	23617408	72688	593903
8	283	21907721	63313	468906
9	282	21883568	68014	454746
10	294	21741568	87389	508279
11	286	21852833	72905	497617
12	265	21944816	96801	532710
13	272	22004636	70959	463033
14	250	22331895	68167	601189
15	264	21968214	70228	496077

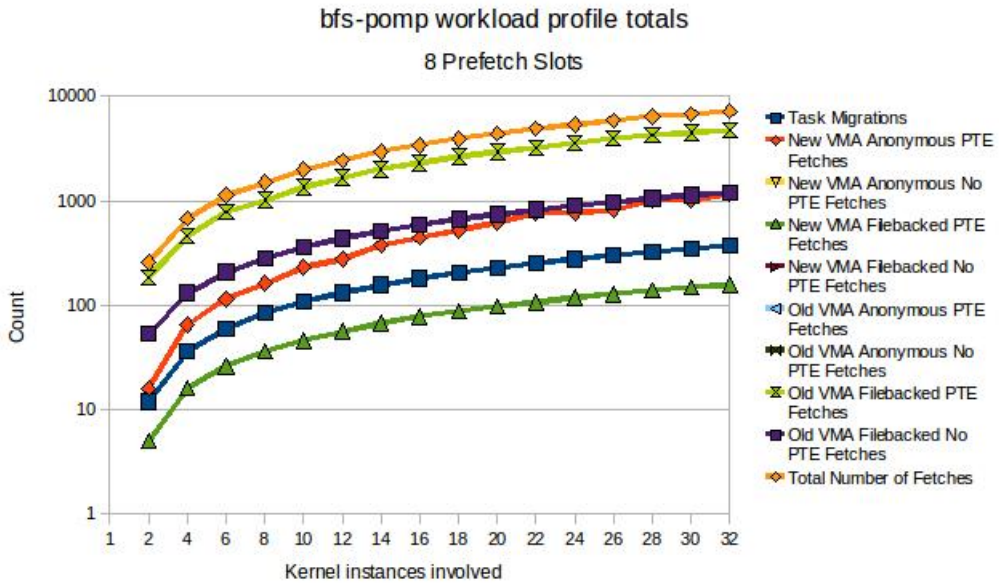

Figure 6.48: BFS Workload Event Totals With 8 Prefetch Slots

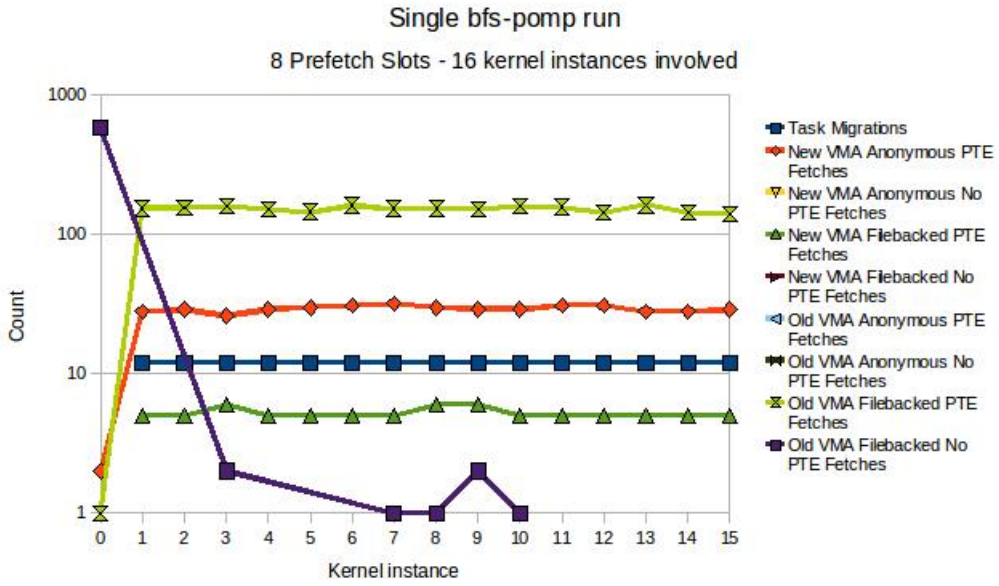

Figure 6.49: BFS Workload profile with 16 kernel instances with 8 prefetch slots

Table 6.16: BFS-POMP fault processing times by kernel instance with 8 prefetch slots

CPU	Count	Max	Min	Average
0	589	18662535	62305	217223
1	187	15839354	83899	546764
2	189	21796486	74381	721281
3	194	22560140	79586	822958
4	185	21789739	82152	614287
5	180	22640701	79991	622517
6	198	21816329	75189	803328
7	191	23187410	100408	812674
8	191	22441040	85864	775074
9	189	34913270	86439	933293
10	195	21755263	87158	681693
11	193	22006642	71898	757791
12	179	22244748	84723	789394
13	196	22370964	84495	641291
14	176	21849980	71111	724050
15	175	22981367	87565	732787

For the average bfs-pomp execution involving 2 kernels, the messaging overhead is 48%, 30%, and 23% for 1 prefetch slot, 4 prefetch slots, and 8 prefetch slots respectively. This percentage decreases due to the increased overhead associated with looking up mappings, and then installing them at their destination.

Table 6.17: BFS-POMP 2 Mapping Retrieval Message Transport Times

	1 Prefetch Slot	4 Prefetch Slots	8 Prefetch Slots
Avg Mapping Fetch Processing (TSC)	41140	73191	96065
Total Message Transit (TSC)	19946	22207	22860
Amount Of Mapping Fetch Processing Waiting For Msg Transit (%)	48.48	30.34	23.79

The perf tool was used to understand which overhead components were most significant in both SMP Linux and Popcorn, and see which ones increased and decreased between the two platforms. These measurements were taken on the main thread of the workload. The greatest

kernel-mode overhead contributors are shown graphed below, as well as the remaining kernel-mode overhead, and the breakdown of time spent in user-mode versus kernel-mode.

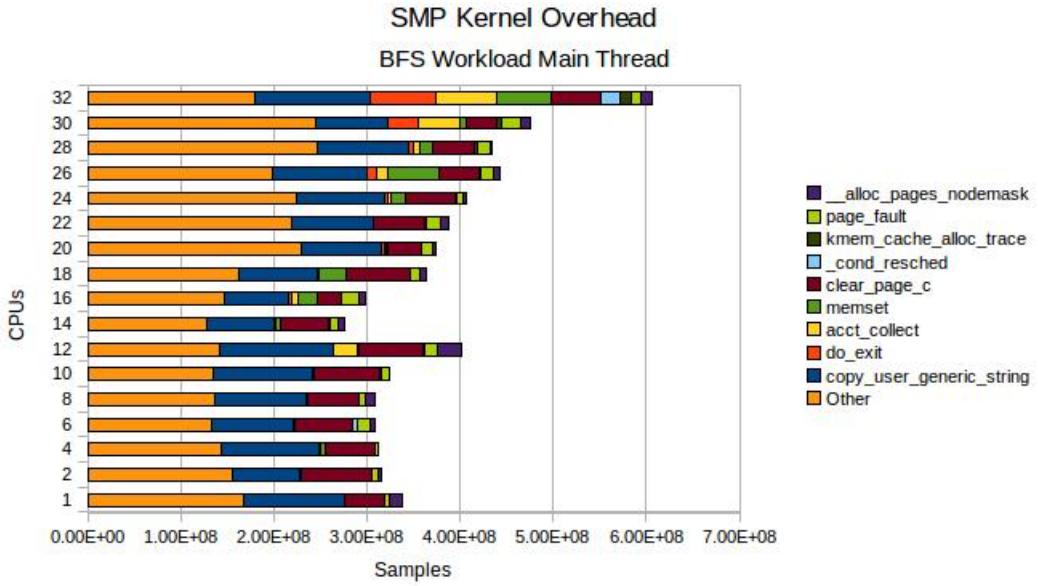

Figure 6.50: Perf Measurement for BFS-POMP Workload on SMP Linux

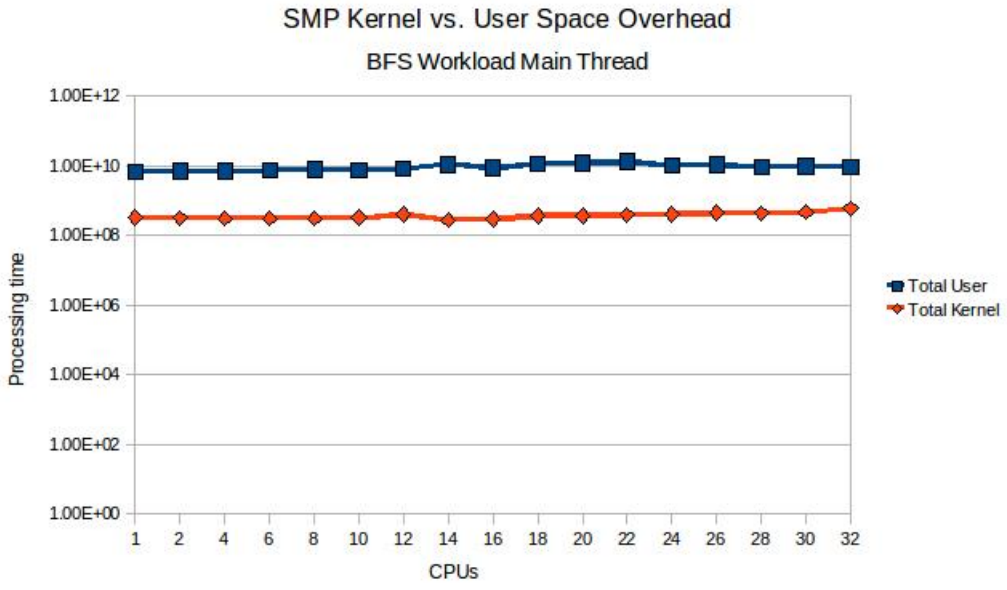

Figure 6.51: Perf Measurement for BFS-POMP Workload on SMP Linux - User vs Kernel

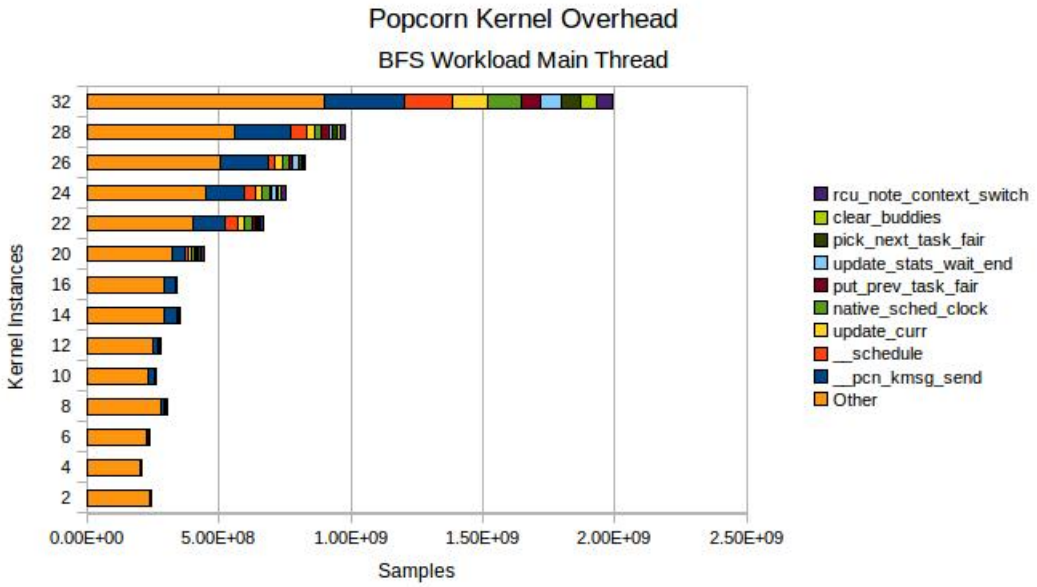

Figure 6.52: Perf Measurement for BFS-POMP Workload on Popcorn

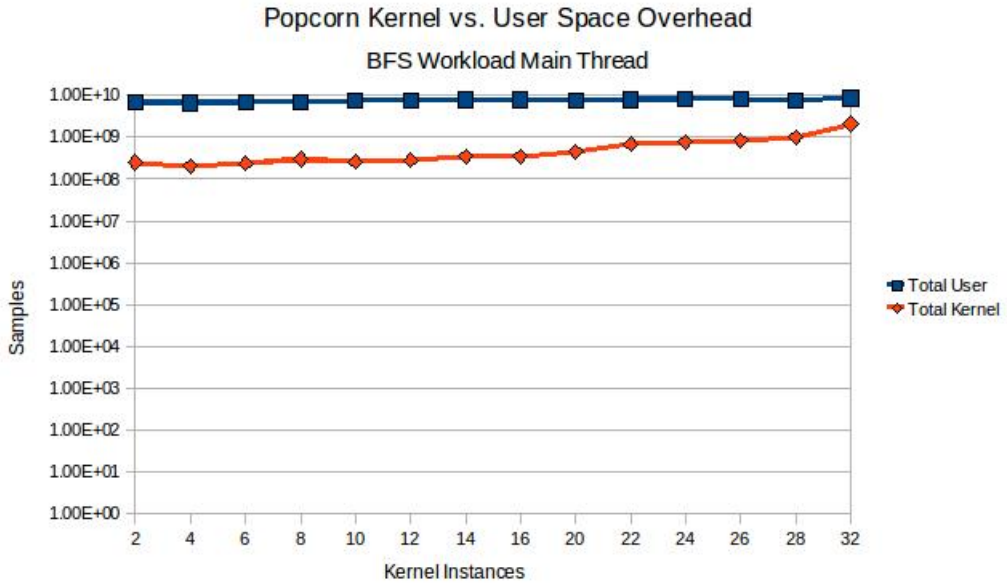

Figure 6.53: Perf Measurement for BFS-POMP Workload on Popcorn - User vs Kernel

The most significant overhead component for SMP Linux is in the function *copy_user_generic_string*, which is a file I/O method. This shows up in the *perf* reading because the measurement is taken on the main thread, and the main thread is responsible for reading the contents of

the file that the BFS workload is processing. The next most significant overhead component is in the function *clear_page_c*, as seen in the previous workloads. These functions are still present in Popcorn, but are replaced in significance by *__pcn_kmsg_send*. This, again, is transport overhead becoming significant at higher kernel instance counts. This overhead is very impactful, given the extremely short duration of the BFS workload.

6.5.2 BFS-POMP Results

Bfs-pomp is extremely interesting for a number of reasons. First, it takes nearly an order of magnitude less time to complete on SMP Linux than any of the other benchmarks. Bfs-pomp takes two orders of magnitude less time to complete on SMP Linux than ft-pomp takes. Because of this, the Popcorn overhead is much more significant, and impacts the workload much greater.

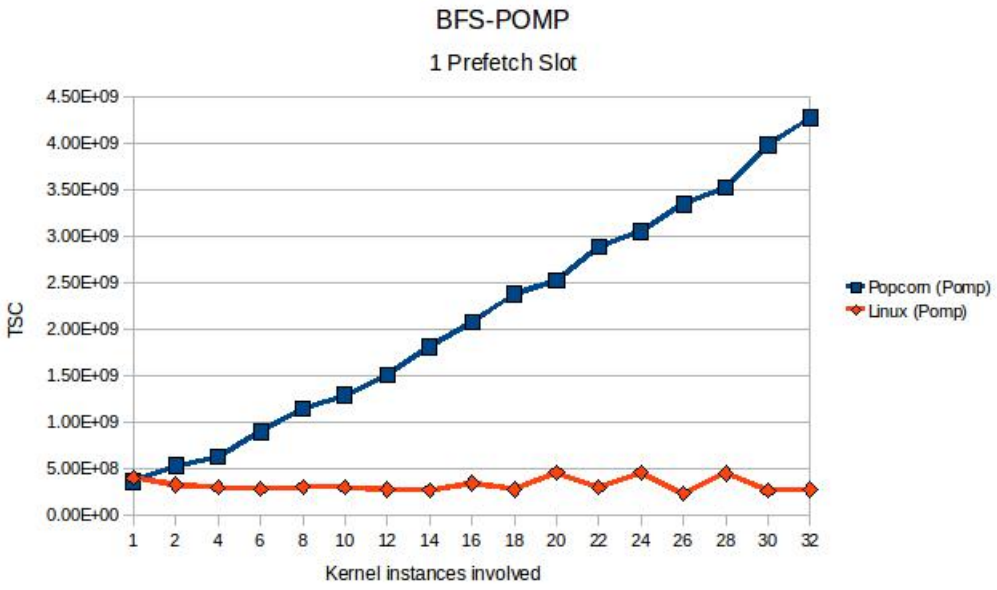

Figure 6.54: bfs-pomp results varying involved kernel instances with 1 prefetch slot

The second reason why bfs-pomp is very interesting is that it sees sizable performance increase when using 4 prefetch slots. However, when using 8 prefetch slots, the performance decreases again. This is due to the fact that Popcorn does not have time to make up for the increase in fetch times because of the workloads short duration. This is an area where further research can be invested. A few suggestions for potential ways to improve this type of workload's performance are in Chapter 8.

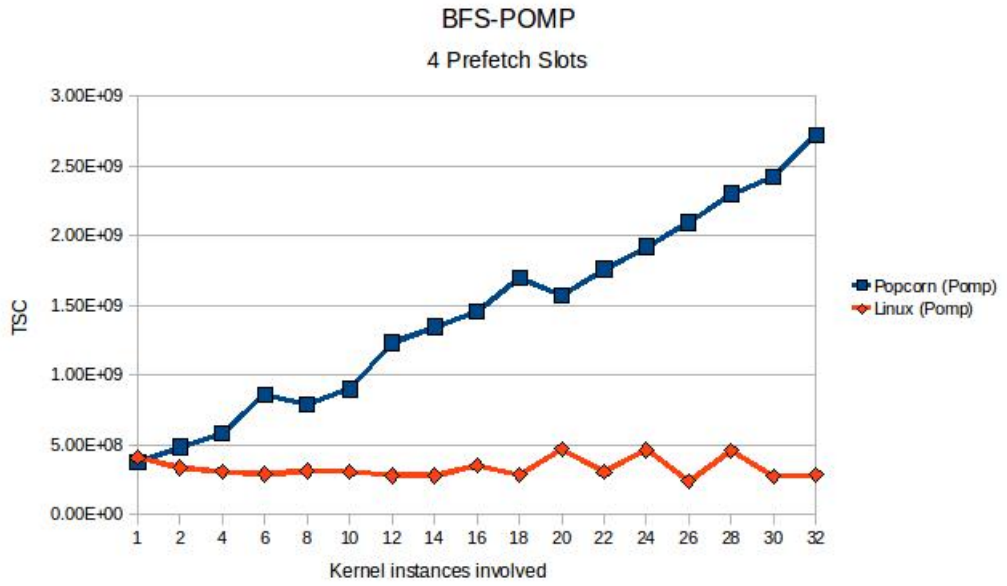

Figure 6.55: bfs-pomp results varying involved kernel instances with 4 prefetch slots

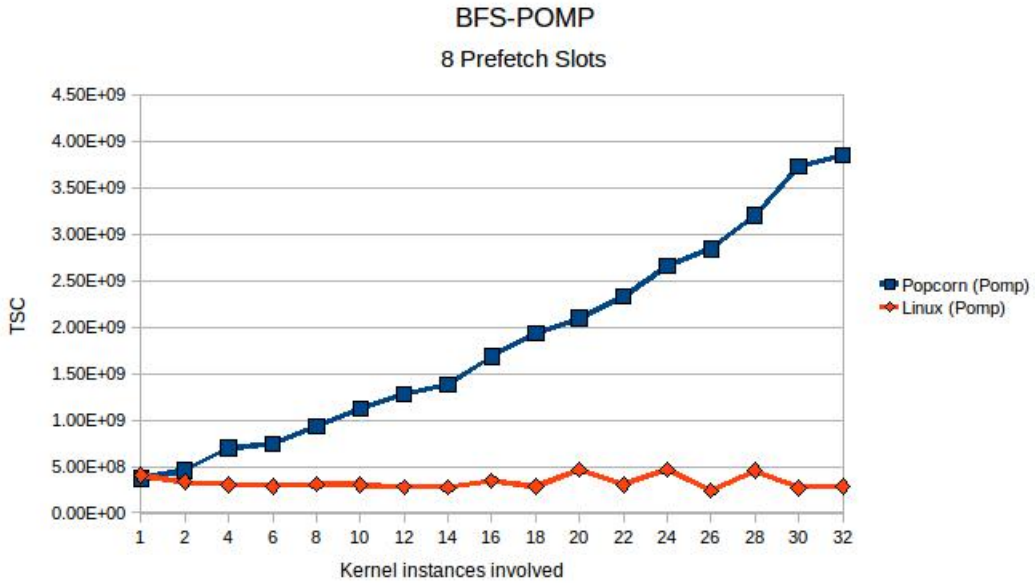

Figure 6.56: bfs-pomp results varying involved kernel instances with 8 prefetch slot

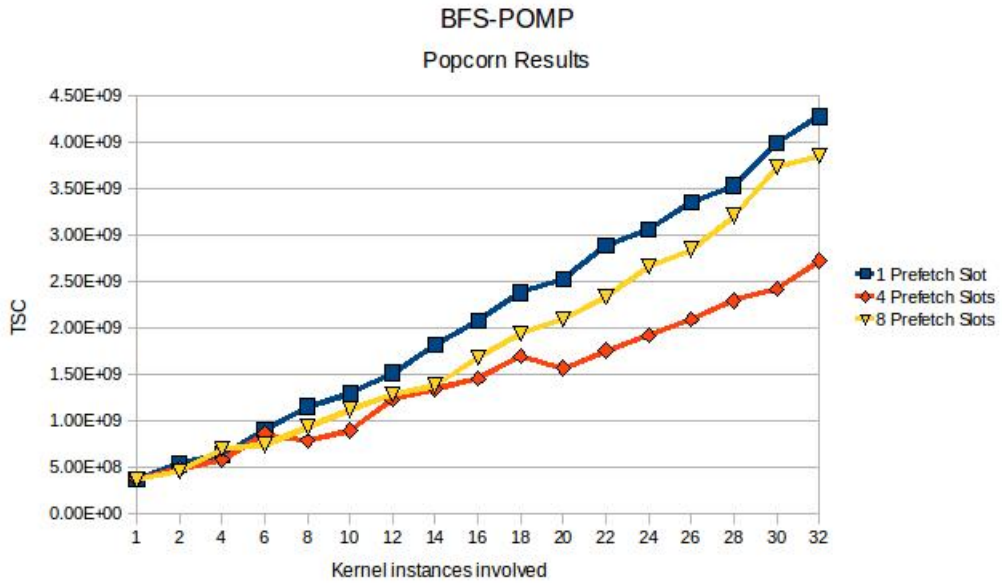

Figure 6.57: bfs-pomp results varying kernel instances involved

SMP Linux scales better than popcorn as the number of kernel instances increase to 32 for the BFS-POMP workload. This is due to the fact that the BFS-POMP workload is so extremely short in duration. Because of this fact, there is no time to make up the gains of the Popcorn overhead in the form of reduced lock contention. However, prefetch is highly effective for this workload, indicating a high degree of shared address space utilization between tasks in the BFS-POMP workload.

6.6 LUD-POMP

The lud-pomp workload is a parallel algorithm for finding the solution to sets of linear equations[9]. Lud-pomp is a Popcorn adaptation of the lud benchmark, which is part of the Rodinia benchmark suite created by UVA. The following section describes the lud-pomp workload in order to give the reader an understanding of what aspects of Popcorn this benchmark exercises most. Once the workload profile is explained, the lud-pomp Popcorn results are presented and are then explained. Popcorn performance is also contrasted with the performance of the same workload on SMP Linux.

6.6.1 LUD-POMP Workload Profile

The lud-pomp workload can be run across as many kernel instances as are present. When the lud-pomp program is invoked, the number of kernel instances to use are specified as a command-line argument. The lud-pomp workload performs one task migration for every kernel instance that is used beyond the first kernel instance. If only one kernel instance is used, no task migrations occur, and all processing is done in the kernel instance on which the program was invoked. The number of migrations for an invocation of lud-pomp involving N kernel instances is therefor N-1.

Figure 6.58 shows the frequency of various significant events as the workload is executed on a varying number of kernel instances. The event types were chosen to show overhead specific to features introduced to the Linux kernel by Popcorn for task migration and address space maintenance. The lud-pomp workload is dominated by mapping fetch operations that successfully result in PTE migrations for file-backed mappings. This type of mapping increases linearly as the number of kernel instances involved increases. Unlike the bfs-pomp workload, the number of file-backed mapping retrieval requests that do not result in PTEs is very low.

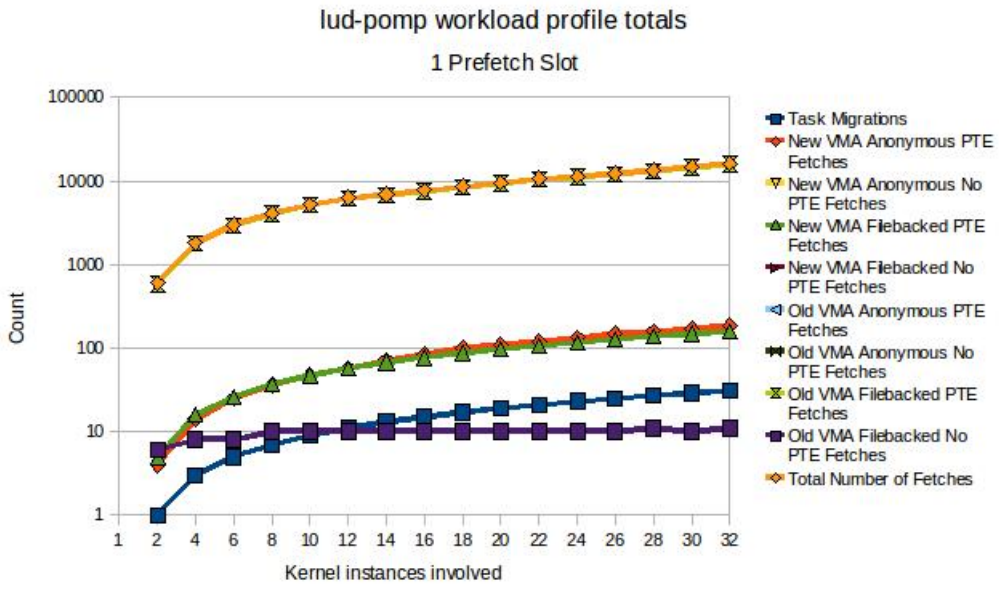

Figure 6.58: LUD Workload Event Totals With 1 Prefetch Slot

Figure 6.59 shows details on how the totals shown above break down per kernel instance for the specific case where 16 kernel instances are involved. This graph shows that every kernel instance except kernel instance zero carries out roughly the same number of mapping retrievals.

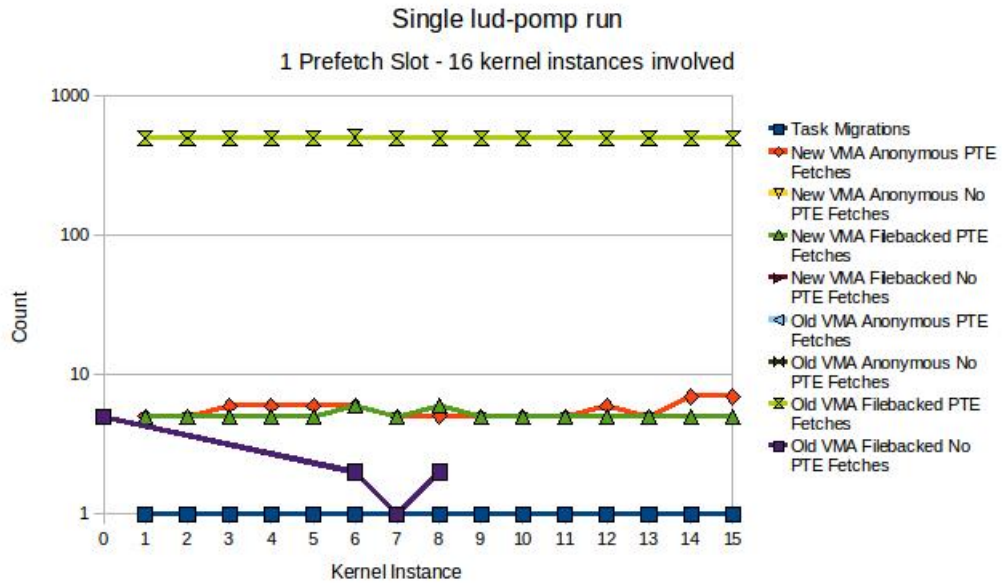

Figure 6.59: LUD Workload profile with 16 kernel instances with 1 prefetch slot

Table 6.18: LUD-POMP fault processing times by kernel instance with 1 prefetch slot

CPU	Count	Max	Min	Average
0	5	83537	69006	73226
1	510	271248	87569	164164
2	511	22963559	85059	256236
3	512	21876390	86479	250133
4	510	23340779	82311	258211
5	510	22052418	81114	266245
6	518	22087033	69796	298354
7	510	25360665	84109	656402
8	515	21879733	79146	339143
9	510	21881530	84489	256790
10	511	21920684	78516	254421
11	509	21787193	81755	520815
12	510	21811266	70942	265816
13	509	22027128	70864	305288
14	512	21813274	66476	276120
15	511	21867902	69981	266958

The following figures show how this workload responds to increasing the amount of prefetch

that is performed during each mapping retrieval. 4 prefetch slot and 8 prefetch slot profiles are shown in these graphs, as opposed to the single prefetch slot that is shown above. The lud-pomp workload is able to reduce its Popcorn overhead incredibly through the use of prefetch. From 1 prefetch slot to 8 prefetch slots, the number of mapping retrievals is reduced by two orders of magnitude.

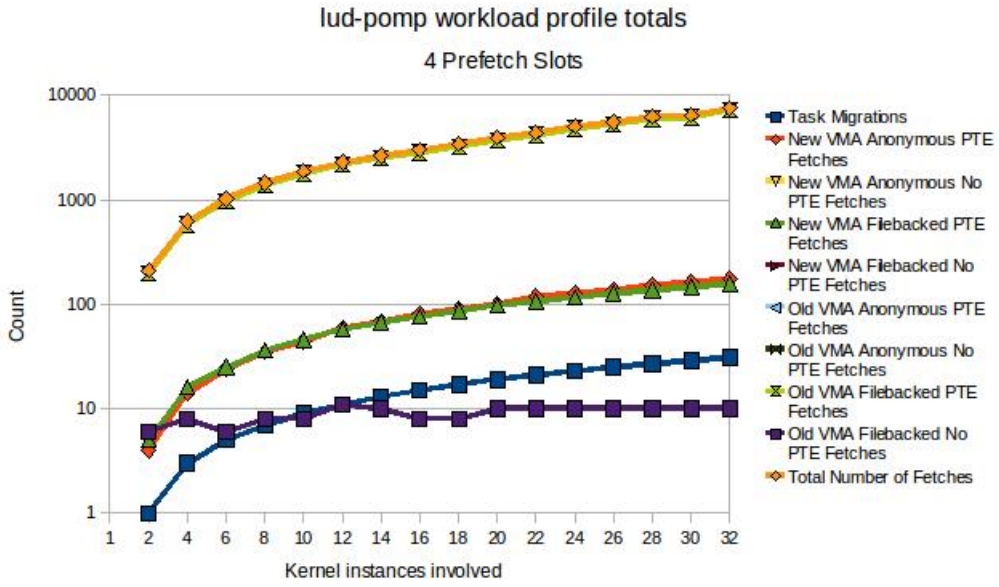

Figure 6.60: LUD Workload Event Totals With 4 Prefetch Slots

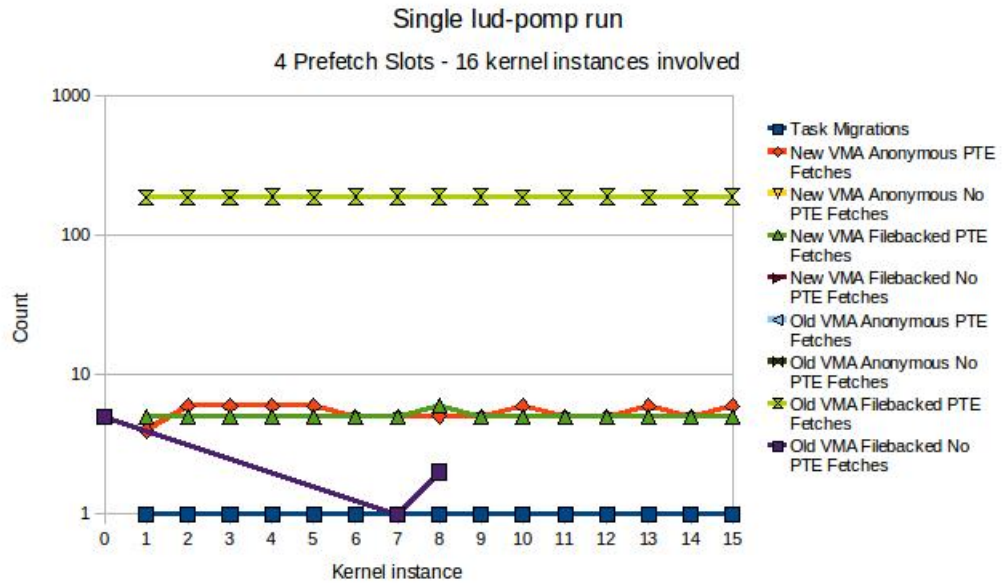

Figure 6.61: LUD Workload profile with 16 kernel instances with 4 prefetch slots

As seen in previous workloads, as the number of fetches decreases, the amount of time required to perform the fetch increases.

Table 6.19: LUD-POMP fault processing times by kernel instance with 4 prefetch slots

CPU	Count	Max	Min	Average
0	5	81462	68436	72479
1	197	11665721	100800	429260
2	199	22530198	83569	386719
3	199	23354606	78869	349165
4	200	22080164	88267	378972
5	199	21908814	89000	320799
6	199	21878606	92923	358102
7	200	23610670	93400	442670
8	202	21810233	102224	460314
9	199	22310014	77038	492384
10	199	21759317	98002	344797
11	198	21848277	89348	419660
12	199	22204938	92977	427373
13	198	21914290	77539	383379
14	198	21798637	82779	486387
15	200	21854477	96961	441179

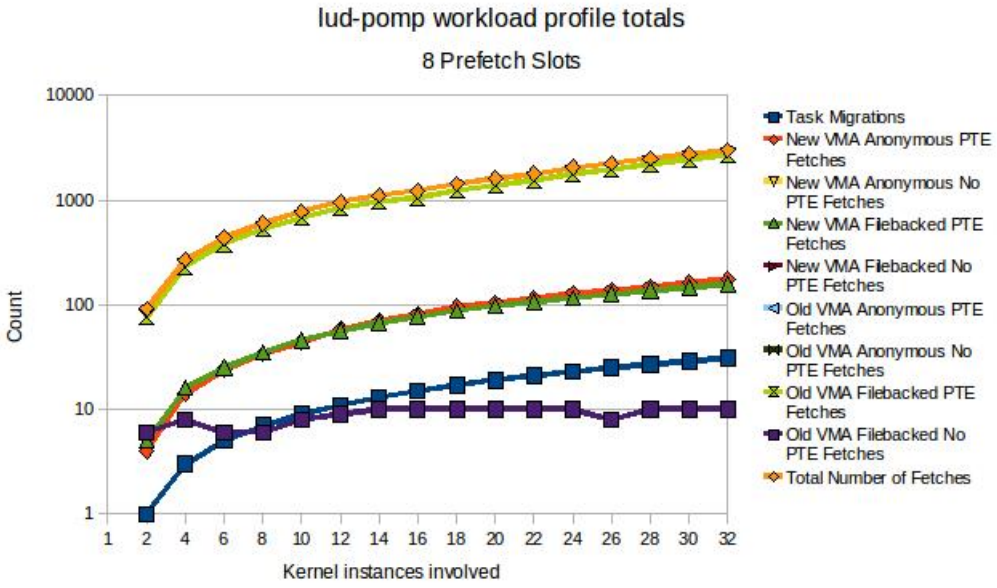

Figure 6.62: LUD Workload Event Totals With 8 Prefetch Slots

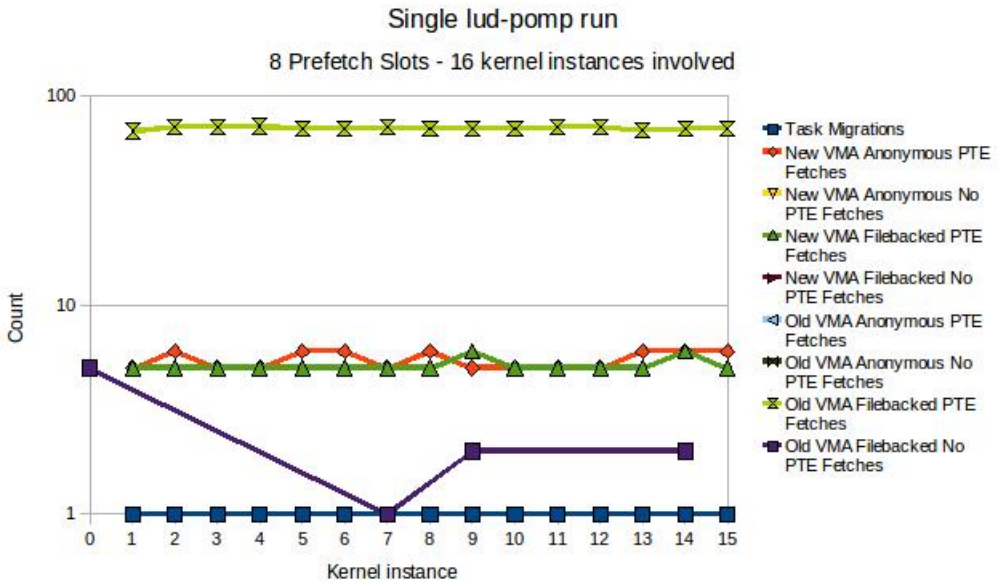

Figure 6.63: LUD Workload profile with 16 kernel instances with 8 prefetch slots

Table 6.20: LUD-POMP fault processing times by kernel instance with 8 prefetch slots

CPU	Count	Max	Min	Average
0	5	81284	67962	72879
1	78	9864214	129440	718704
2	82	21781131	125979	970687
3	81	21946207	121243	944141
4	82	22059629	119035	832517
5	81	21815207	110772	898268
6	81	21987477	98886	908398
7	82	24412068	134514	1188549
8	81	22261237	121301	918439
9	83	21775727	102335	1078789
10	80	22428139	124203	921261
11	81	21873153	143405	903863
12	81	22177350	119112	933536
13	80	22253744	135080	1204257
14	84	22884140	130555	1260517
15	81	22951512	124432	1094660

For the average lud-pomp execution involving 2 kernels, the messaging overhead is 51%,

39%, and 28% for 1 prefetch slot, 4 prefetch slots, and 8 prefetch slots respectively. This percentage decreases due to the increased overhead associated with looking up mappings, and then installing them at their destination.

Table 6.21: LUD-POMP 2 Mapping Retrieval Message Transport Times

	1 Prefetch Slot	4 Prefetch Slots	8 Prefetch Slots
Avg Mapping Fetch Processing (TSC)	45266	58428	86505
Total Message Transit (TSC)	23280	23193	24476
Amount Of Mapping Fetch Processing Waiting For Msg Transit (%)	51.42	39.69	28.29

The perf tool was used to understand which overhead components were most significant in both SMP Linux and Popcorn, and see which ones increased and decreased between the two platforms. These measurements were taken on the main thread of the workload. The greatest kernel-mode overhead contributors are shown graphed below, as well as the remaining kernel-mode overhead, and the breakdown of time spent in user-mode versus kernel-mode.

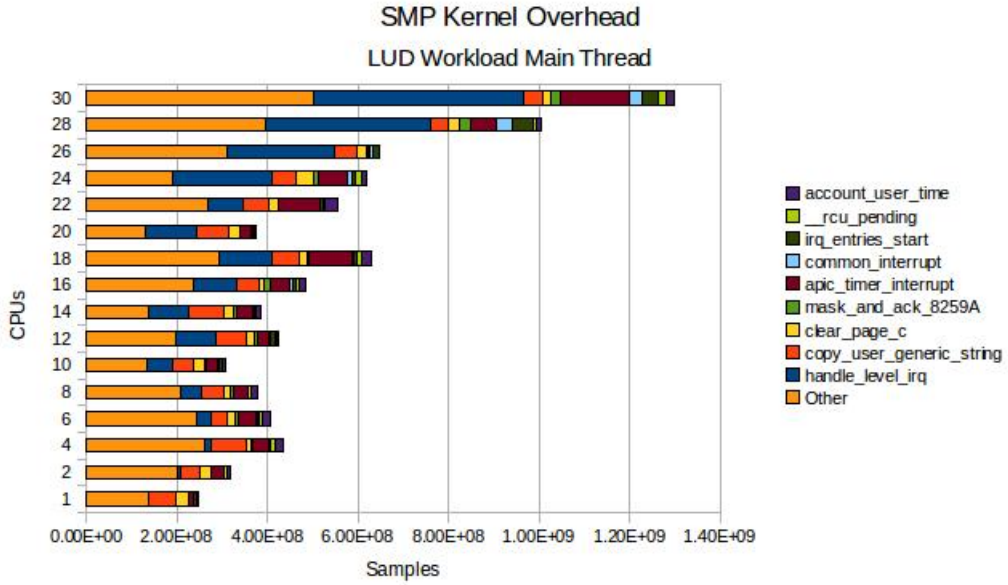

Figure 6.64: Perf Measurement for LUD-POMP Workload on SMP Linux

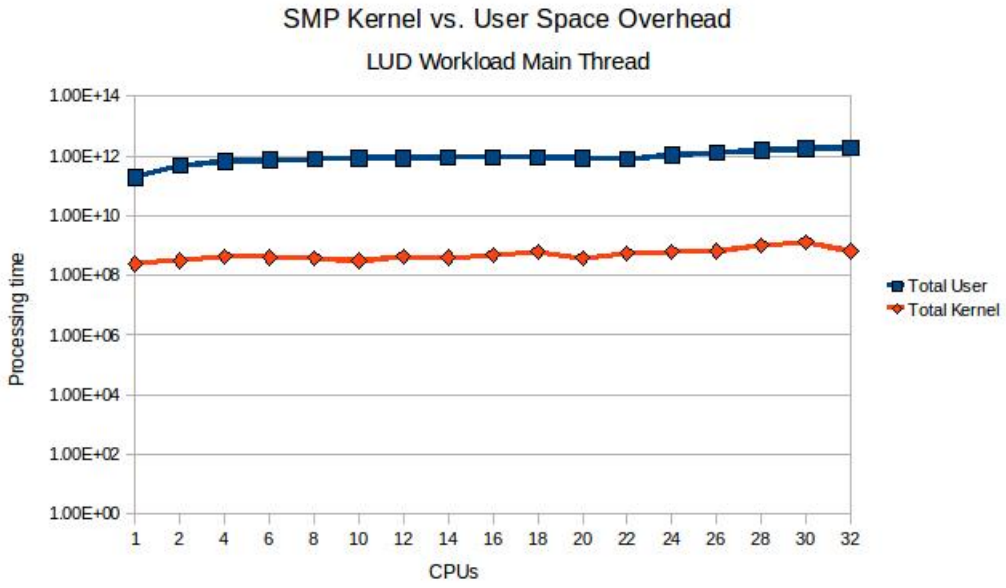

Figure 6.65: Perf Measurement for LUD-POMP Workload on SMP Linux - User vs Kernel

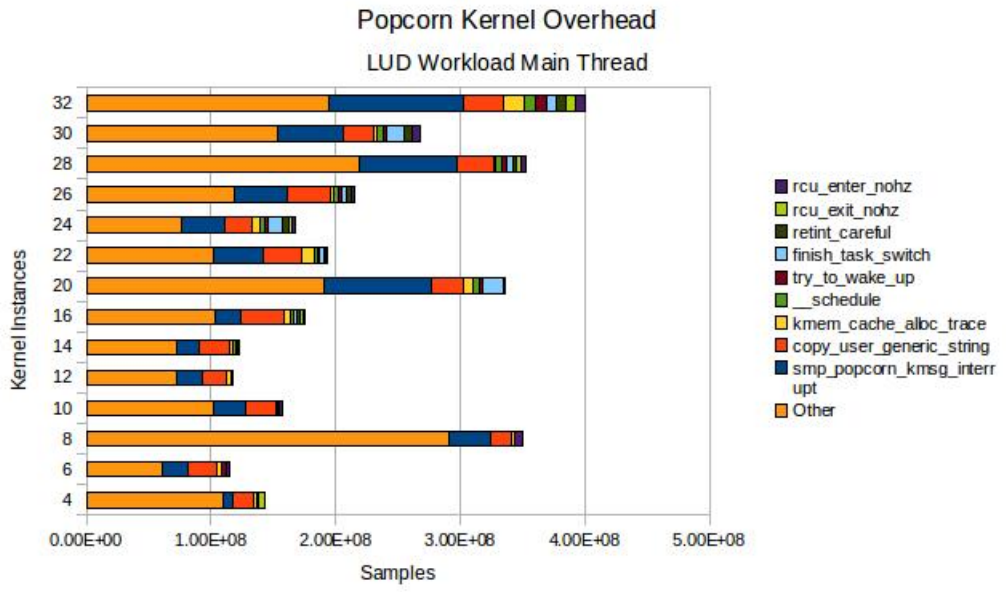

Figure 6.66: Perf Measurement for LUD-POMP Workload on Popcorn

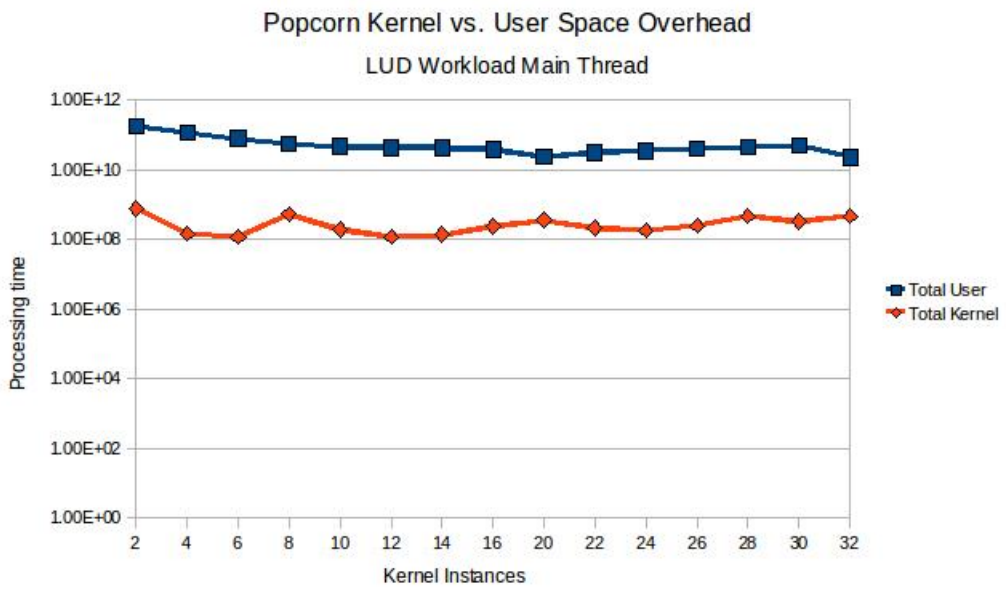

Figure 6.67: Perf Measurement for LUD-POMP Workload on Popcorn - User vs Kernel

The SMP LUD workload is increasingly dominated by time spent in a spin lock in *handle_level_irq*, and *apic_timer_interrupt*, due to timer related APIC interrupt coupling between SMP nodes. These types of overhead are not seen in significant amounts in Popcorn, but are instead replaced in significance by messaging overhead that increases with kernel instance count.

6.6.2 LUD-POMP Results

The Popcorn lud-pomp results track very closely with SMP Linux. The Popcorn overheads are lost in the noise of this extremely long duration workload. Little appreciable performance degradation or gain is apparent as a result of the sizable decrease in mapping retrieval operations as prefetch is increased. This also is attributable to the very long duration of the lud-pomp workload, which allows the decrease in scheduling overhead to compensate for the task migration and address space consistency maintenance overhead. However, the trend noted above with respect to increases in prefetch not always yielding performance gain is visible in this workload. Figure 6.71 shows a direct comparison between all of the Popcorn lud-pomp data. It indicates that when 4 prefetch slots are used, some gain is made over the cases where 1 and 8 prefetch slots are used.

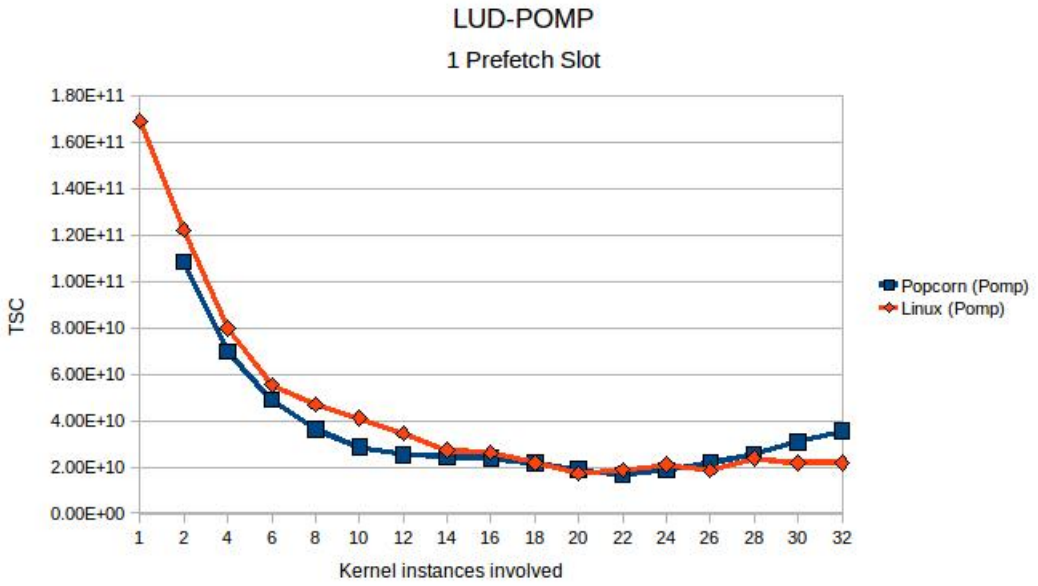

Figure 6.68: lud-pomp results varying involved kernel instances with 1 prefetch slot

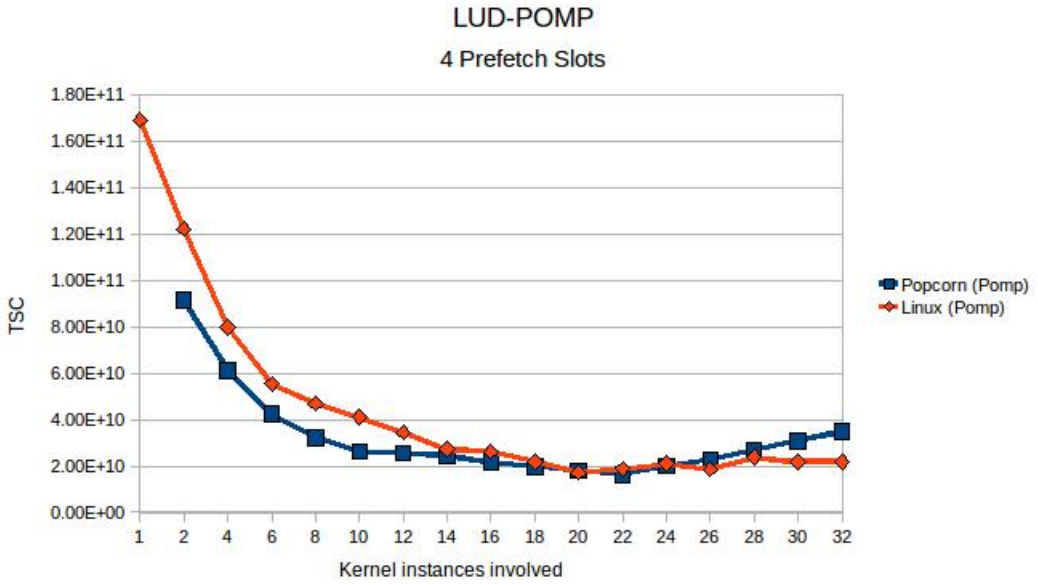

Figure 6.69: lud-pomp results varying involved kernel instances with 4 prefetch slots

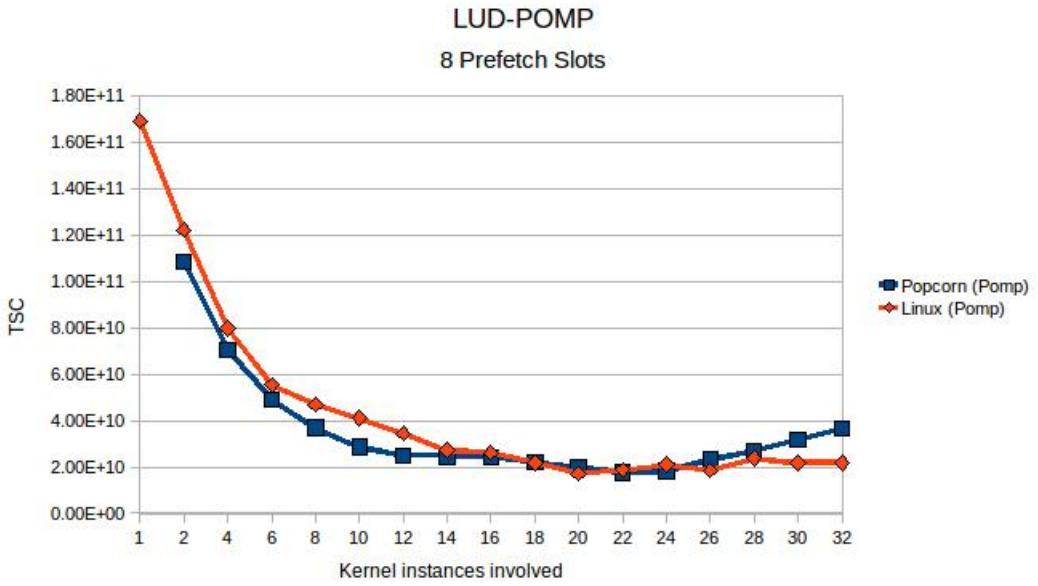

Figure 6.70: lud-pomp results varying involved kernel instances with 8 prefetch slot

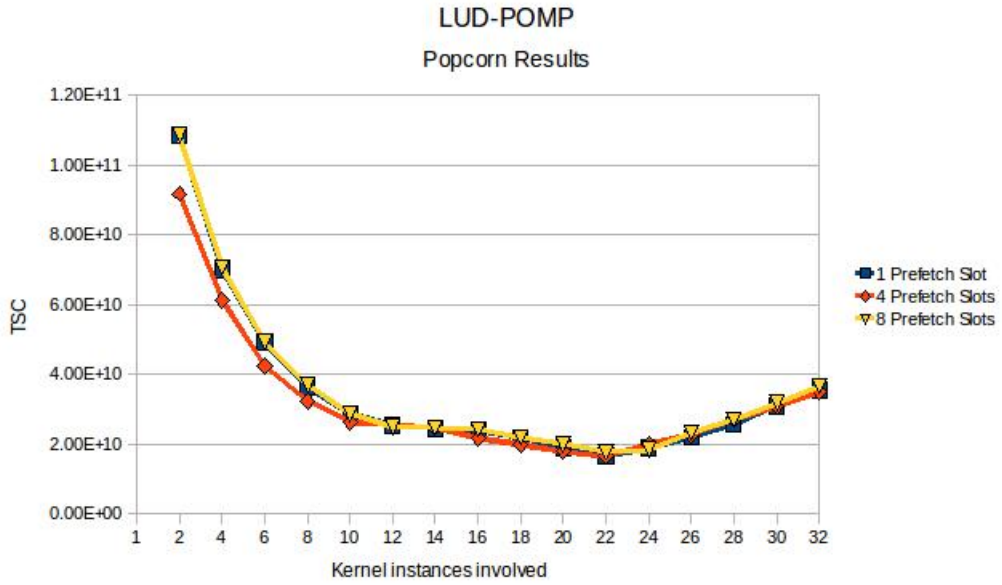

Figure 6.71: lud-pomp results varying kernel instances involved

Popcorn scales better than SMP Linux for the first 22 kernel instances, but is then slightly outpaced by SMP Linux through 32 kernel instances. The LUD-POMP workload is unique in that it has relatively low mapping migration requirements, and yet the workload is of

very long duration. Because of that, it is able to realize sizable gains as lock contention is reduced relative to SMP Linux. However, the Popcorn messaging system scales poorly, and as a result the high messaging cost exceeds the gains at higher kernel instance counts.

6.7 Benchmark Discussion

The benchmarks that were chosen for workloads demonstrated many aspects of this project, and while performance is not the primary objective of this work, a number of interesting performance related characteristics have emerged from their execution that will now be discussed.

6.7.1 SMP Linux versus Popcorn Linux

Popcorn Linux is fundamentally different than SMP Linux in its processing profile. Popcorn Linux has decoupled kernel instances, and therefore removes most of the contention for locks within the memory management subsystem. SMP Linux, on the other hand must pay for the lock contention since every CPU it uses must synchronize. In Popcorn, the effect of the lack of contention in the memory subsystem battles the effect of the performance overhead increase due to messaging, which is discussed in the following section.

6.7.2 Mapping Retrieval Overhead and Symmetry

The is-pomp, ft-pomp, and lud-pomp on Popcorn all performed very similarly to SMP Linux. The bfs-pomp and cg-pomp workloads appear to display higher sensitivity to the Popcorn overhead.

The cg-pomp workload is limited by the huge asymmetry in its mapping fetch overhead. In that workload, kernel instance 0 is a processing bottleneck. It is responsible for loading the vast majority of pages from disk, where all other kernel instances load only a few. The pages that the other kernel instances work on are migrated from kernel instance 0. Kernel instance 0 spends much of its time asking other kernel instances if they have pages for file backed mappings, receiving negative responses, and then loading those pages from disk. Additionally, because cg-pomp is overwhelmingly dominated by mapping requests that receive negative responses, much of the work that is done by popcorn to acquire locks and carry out queries is wasted. Removing this unnecessary overhead is difficult because no mechanism exists for determining whether other kernel instances have mapped regions without querying them each time.

The bfs-pomp workload suffers from the same problem seen in cg-pomp in that many mapping queries result in negative results. This problem is greatly exacerbated by the fact

that its execution has an extremely short duration in time. It makes sense to compare the bfs-pomp workload with the lud-pomp workload, since its fetch profile is very similar, being dominated by file-backed page migrations accompanied by PTEs. What sets those two benchmark workloads apart is that it takes nearly 3 orders of magnitude less time to execute bfs-pomp on SMP Linux than it does to execute lud-pomp on SMP Linux. While the duration of the workload is decreased by orders of magnitude, the number of total faults observed in Popcorn is decreased by only half between lud-pomp and bfs-pomp. This means that the Popcorn overhead is significantly more pronounced, and that is seen in the bfs-pomp result graphs. This implies a link between execution time of a workload, the number of mapping fetches that are required to support it, and performance.

6.7.3 Prefetch

It was assumed at the beginning of this effort that prefetch would have a profound effect on workload performance for workloads that share many pages among thread group members. This was true to an extent. All workloads that shared many pages among thread group members did experience drastic reductions in the number of page retrievals as the amount of prefetch increased. This is due to the fact that increased prefetch obviated the need to perform many future retrievals, since faults for the prefetched memory areas no longer occurred. However, this only had very significant effects on the overall performance of benchmark workloads, such as bfs-pomp, that are short in duration. Workloads that are longer in duration did not show great variation due to prefetch, as in those other workloads, the Popcorn overhead did not dominate over the nominal benchmark processing as it did in bfs-pomp.

Another very interesting characteristic of the prefetch results is that a linear increase in prefetch amounts does not yield a linear improvement in workload performance. This is attributable to a number of effects. As prefetch increases, the number of faults that must be serviced decreases. This is a gain for the system, since there is work associated with the mapping retrievals that are triggered by faults. However, there are forces that work counter to this positive effect at the same time. As prefetch increases, the amount of time spent by a kernel instance responding to a mapping retrieval request to fill prefetch slots increases as well. This takes time. Additionally, as more prefetch information is provided to a querying kernel instance, the amount of time that is needs to install the prefetched mappings increases as well. Lastly, as the amount of prefetch increases, the size of the messages increases as well. Due to the method that the Popcorn messaging layer uses to chunk large messages, this increases the amount of time necessary to transport the mapping response messages, which carry the prefetch data.

6.7.4 Transport Speed

A lot of work must be done by Popcorn in order to maintain a consistent address space across a distributed thread group. Most of the overhead introduced by this is in the form of performing mapping fetches. These fetch actions are comprised of first acquiring the Lamport lock, performing the fetch, then releasing the lock. This requires 5(N-1) messages, where N is the number of kernel instances involved in the workload. The measurements that were made above reveal that if only the fetch operation is considered, up to 60% of the overhead may be spent waiting for the transport to deliver messages from one kernel instance to another. This is highly significant. If Lamport lock acquisition and release were included, and assuming the lock were acquired without contention, that percentage would likely rise, as the work associated the Lamport locks involves significantly less processing than the mapping fetch and installation process, and therefore the ratio of time spent in the transport relative to other processing would increase.

Chapter 7

Conclusions

In this thesis Popcorn Linux was extended to support task and address space migration across kernel instances in a homogeneous x86 64bit environment, supporting both process and thread migration. Address space consistency for distributed thread groups was implemented and shown to be correct. These mechanisms were proven to work through design review, and extensive testing. Though performance was not the driving goal of this project, performance measurements were taken, and evaluated under five different workload profiles.

It was found that Popcorn reduces its overhead by removing lock contention within the memory subsystem and interrupt management subsystem. At the same time, it increases its overhead by messaging between kernel instances. These two forces act against one another, and depending on the workload and number of active kernel instances, Popcorn may or may not perform well when compared to SMP Linux.

It was found that most of the overhead associated with task and address space migration is attributed to address space migration. Much of that time was spent in the transport layer, moving requests and responses between kernel instances. This could be reduced in the future by 1) implementing a more efficient transport, and 2) reducing demand for messaging in the protocols that use messaging. Suggestions for how to accomplish the latter are presented in Chapter 8.

Another significant observation related to performance is that the duration of the process relative to the mapping retrieval overhead matters. Workloads that are extremely short in duration do not benefit much from migration in cases where large parts of the address space must be utilized by the migrated task. This is due to the fact that mapping retrievals must occur in order to provide access to the utilized regions of the address space. Each of these retrieval operations incurs an overhead cost. So the fewer fetches that are necessary, the higher the migration utility.

Additionally, it was found that prefetch has significant benefit for workloads that do mapping retrievals that result in successfully finding existing virtual to physical mappings. However,

for workloads that perform mapping retrievals that do not successfully find existing virtual to physical mappings, increased prefetch unnecessarily results in increased overhead. Chapter 8 recommends methods of dynamically adapting Popcorn's mapping retrieval protocols to workload needs in order to utilize prefetch only in situations where it helps.

Chapter 8

Future Work

8.1 Identify and Resolve Performance Bottlenecks

Performance optimization should be continued in order to find and eliminate performance bottlenecks. Many optimizations have already been made in the course of this thesis work, but there are undoubtedly remaining bottlenecks.

One ripe area for investigation is the distributed locking mechanism. Alternatives to Lamport's distributed mutual exclusion algorithm can be explored, potentially reducing the messaging requirements for address space consistency.

The above recommendation aims to reduce the messaging overhead by reducing demand for messaging. It is also recommended that the messaging layer be re-visited to explore alternative implementations with the goal of reducing the overhead of all messaging-heavy components within Popcorn. It was found during this work that the messaging layer incurs a significant overhead, accounting for up to 60% of the mapping retrieval overhead. This overhead will likely continue to increase as Popcorn approaches its goal of providing a single system image, as an increasing number of kernel components will utilize the messaging layer. Effort aimed at reducing all messaging overhead would benefit all current and future users of the messaging layer.

8.2 Alternative Mapping Retrieval Protocols

The protocol for distributed address space consistency and availability that was adopted in this work involves the migration of mappings as those mappings are needed. When a fault occurs, the faulting kernel instance asks all other kernel instances for a mapping for the virtual page that caused the fault. Each of the kernel instances that receives this request

inspects its memory map for the faulting thread group for a mapping that would satisfy the request, and responds accordingly. The faulting kernel instance is responsible for assembling the responses to arrive at a final determination of which response to use. This is not the only possible way to orchestrate mapping migration. Because the vast majority of the overhead associated with task migration is due to maintaining a consistent address space, investigation into alternative and more performance optimized mechanisms for mapping migration is advised.

It is not necessary to select one specific a mapping retrieval protocol for exclusive use by all distributed tasks. A protocol may instead be selected from the implemented protocols at run-time adaptively to select the optimal mapping retrieval protocol for the workload at the current time. If that mapping retrieval fails to perform, based on actively maintained metrics such as fault rate, the mechanism can be abandoned in favor of another one. Additionally, the mapping retrieval protocol employed for a given task at a given time need not be the same between tasks and processes. The mapping retrieval protocol could be selected each time a retrieval is initiated to optimize the workloads performance and the systems performance based on metrics maintained on a per task, per process, per kernel instance, or per cluster basis. A set of recommended metrics to maintain and use in making a determination about which mode of mapping retrieval to use includes:

- The rate of faults each task is experiencing
- The type and quantity of mappings that are being retrieved when a retrieval is made
- The number of pages that are prefetched each time a retrieval is done

8.2.1 Single Query Single Response Mapping Query

One alternate mechanism for acquiring mappings involves sending the request to only one other kernel instance. That kernel instance would look for a mapping in its local memory map for the faulting distributed thread group. If a complete mapping is found that includes both a VMA and a PTE for the faulting address, it responds directly to the faulting kernel instance. If no mapping is found, or only a VMA is found, it bundles up its findings, and forwards that information along with the original request to the next kernel instance. The next kernel instance then repeats that process, and makes a decision about whether its findings represent a more complete response than what was previously found. If it finds a complete mapping, it responds directly to the faulting kernel. If not, it again forwards the current mapping response to the next kernel instance. This pattern repeats until either a complete mapping response has been arrived at, or all kernel instances have been asked for a mapping and none have found one. At the end of this process the faulting kernel instance receives a single, complete response. That response needs no further vetting and can be used as is.

A subtle twist on this algorithm includes intelligently selecting the order in which the request is routed through kernel instances. One possible routing protocol might involve prioritizing kernel instances that have previously yielded good mapping responses. Another might involve routing first to kernel instances that are known to host tasks that are in the same distributed thread group.

A variation of this algorithm forces the query to make its way through all active kernel instance regardless of whether or not the query has been satisfied before the response is sent back to the faulting kernel instance. This allows the mapping to be imported into all kernel instances as its being forwarded through all kernel instances. This is effectively a prefetch mechanism, and would benefit distributed workloads that share many pages. In this variations, optimizations could be made to avoid doing resource intensive computing when the system is under high load. For example, when a kernel instance receives a mapping request and that kernel instance is not the requesting kernel instance, it could first look to see if that request has already been satisfied by another kernel instance. If it has, and the current kernel instance is under high load, it might choose to skip installing the mappings locally, and instead forward the request to the next kernel instance. Whether or not this is an optimization depends on whether or not the mapping that has been resolved will be needed by the kernel instance that chose not to install the mapping.

This alternate mapping retrieval protocol might be useful in systems with high messaging latency. This is due to the fact that the number of messages that must be sent in this algorithm have an upper bound of the number of kernel instances, whereas the currently implemented algorithm involves sending two times that number. This alternate protocol might also be useful in high load scenarios where taking all kernel instances off-line for the time necessary to perform the mapping lookup is constraining. There are, however, also potential down sides to utilizing this alternate protocol. The amount of time necessary to perform a mapping lookup would increase due to the serialized nature of the lookup message forwarding process. It is unclear whether the benefits outweigh the drawbacks to this alternate algorithm. Further research into these trade offs would be useful to determine which types of workloads would benefit from employing which mapping acquisition methods.

8.2.2 Distributed Thread Grouping Based Mapping Query

Another potential protocol for acquiring mappings involves exploiting knowledge of which other kernel instances have in the past or are currently hosting tasks in the faulting task's distributed thread group. Given this knowledge, a kernel instance could poll only those remote kernel instances. This minimizes overhead imposed on kernel instances that have never hosted members of this distributed thread group. This method requires a solid mechanism for tracking which kernel instances have hosted members of a given distributed thread group. This can be accomplished in a number of different ways. For example, perhaps when a task migration occurs, the receiving kernel instance broadcasts a notification to all other

kernel instances to signal its participation in the task's distributed thread group. When other kernel instances receive this broadcast, they make a note of this fact and then respond with an acknowledgment message. Once all acknowledgments have been received, the newly hosting kernel instance can resume the execution of the migrated task. This ensures that all kernel instances always have complete knowledge of which kernel instances participate in which distributed thread groups, forming the basis of a more streamlined mapping request mechanism.

This approach reduces processing overhead for kernel instances that are not involved in processing on behalf of a given distributed thread group. It also removes messaging overhead associated with communicating with those uninvolved kernel instances.

8.3 Shared File System

Currently, each kernel instance has its own independent file system. This greatly limits the types of meaningful workloads that can be executed on the Popcorn platform, as threads running on different kernel instances cannot read to or write from shared files. Introducing support for a single cross kernel instance file system would drastically improve the usability of the Popcorn system.

8.4 User-Space Locking Abstractions

Many common user-space libraries depend on locking abstractions, including futex, semaphore, and mutex, for execution ordering and data guarding purposes. While these abstractions are supported within a kernel instance, they are currently not supported across kernel instances. Work must be done to provide this functionality to maintain POSIX compliance and extend support to the huge body applications that require that compliance.

8.5 Scheduling Support

Support for load balancing across kernel instances is not currently integrated into the Popcorn scheduling mechanism. A kernel instance will currently only initiate a task migration upon receiving explicit instructions from user space to do so. This is a very interesting area for further research, as the decision to do a migration would optimally take into account the benefit of moving the workload, balanced with the overhead associated with not only moving the task itself, but also with moving mappings and maintaining address space consistency. In the case where the system has no knowledge of which virtual pages a given task will access, this could be done stochastically using the task's address space migration history as data.

In contrast, given knowledge of which virtual pages a task will use in the next scheduling cycle, this calculus could be made more precise. This would be possible with linker support where the linker embeds address space usage information into binaries as metadata that the scheduler could consult to make intelligent migration decisions.

Bibliography

[1] D. Aloni. Cooperative linux. *Proceedings of the Linux Symposium*, July.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. *RNR Technical Report RNR-94-007*, 1994.

[3] A. Baumann, P. Barham, P Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. *22nd Symposium on Operating Systems Principles*, 2009.

[4] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2006.

[5] S. Boyd-Wickizer, H. Chan, 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. *OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[6] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N Zeldovich. An analysis of linux scalability to many cores. *OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 1997.

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. *ACM Symposium on Operating Systems Principles*, December 1995.

[9] S. Chi, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, December 2010.

[10] A. Clements, M. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[11] A. Clements, M. Kaashoek, and N. Zeldovich. Radixvm: scalable address spaces for multithreaded applications. *EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[12] J. Colmenares, G. Eads, Hofmeyr. S., S. Bird, M. Moreto, D. Chou, B. Gluzman, E. Roman, D. Bartolini, N. Mor, K. Asanovic, and J. Kubiatowicz. Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. *ACM*, May 2013.

[13] Wikipedia contributors. Msm7000. http://en.wikipedia.org/wiki/MSM7000, September 2013.

[14] Wikipedia contributors. Arm big.little. http://en.wikipedia.org/wiki/ARM_big.LITTLE, 2014. Accessed: 2014-07-20.

[15] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *SOFTWAREPRACTICE AND EXPERIENCE, VOL. 21(8), 757785*, August 1991.

[16] B. Gamsa. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[17] Intel. Multiprocessor specification, 1997.

[18] T. Jones. Kernel apis, part 1: Invoking user-space applications from the kernel, February 2010.

[19] A. Joshi, P. Swapnil, M. Naik, S. Rathi, and K. Pawar. Twin-linux: Running independent linux kernels simultaneously on separate cores of a multicore system. 2010.

[20] E. Keller, J. Szefer, J. Rexford, and R. Lee. Nohype: Virtualized cloud infrastructure without the virtualization. *ISCA*, June 2010.

[21] J. Moreira, M. Brutman, J. Castanos, T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B Wallenfelt. Designing a highly-scalable operating system: The blue genełstory. November 2006.

[22] T. Shimosawa and Y. Matsuba, H. Ishikawa. Logical partitioning without architectural support. *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, July 2008.

[23] D. Wentzlaff and A. Agarwal. The case for a factored operating system (fos). 2008.