MEMORY REGION: A SYSTEM ABSTRACTION FOR MANAGING THE COMPLEX MEMORY STRUCTURES OF MULTICORE PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Min Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2013

MEMORY REGION: A SYSTEM ABSTRACTION FOR MANAGING THE COMPLEX MEMORY STRUCTURES OF MULTICORE PLATFORMS

This dissertation is dedicated to my wife, Hyunsuk Kim.

ACKNOWLEDGEMENTS

It has been a great privilege to spend several years at the Georgia Institute of Technology. I still remember the day when I arrived at the Atlanta airport, but now it's already time to leave the school. Atlanta has become a special city to me, where I met my wife, started my Ph.D. program, and where my son was born. I will always miss Atlanta, and Georgia Tech, and all the good days I have spent there.

It was my pleasure to meet all of my great friends and fellow students – Minsung Jang, Minjang Kim, Jingu Kim, Vishakha Gupta, Mukil Kesavan, Vishal Gupta, Priyanka Tembey, Hrishikesh Amur, Chengwei Wang, Fang Zheng, Alex Merritt, and many others.

I'd like to thank my committee members, Professor Sudhakar Yalamanchili, Professor Ada Gavrilovska, Professor Sham Navathe, and Dr. Rob F Van Der Wijngaart for all of their help. I have been fortunate to work with them. Special thanks to Dr. Rob F Van Der Wijngaart for his warm support and assistance.

Above all, I would like to express my sincere gratitude to my advisor Prof. Karsten Schwan for his continuous support of my Ph.D. studies and research. He has always supported me and guided me whenever I was struggling with various problems. My idea about the memory-centric scheduler has been guided by him, and he always helped me understand the systems perspective in pursuing it. Thank you for all for your support.

Also I thank Rob Knauerhase at Intel for accepting me as an intern. I thank A. S. Krishnakumar, P. Krishnan, Navjot Singh, Shalini Yajnik at Avaya Labs, as well. It was my privilege to be able to work with you.

I would never have been able to finish my dissertation without the guidance of my advisor, Karsten Schwan, help from my friends, and support from my family and wife. I would like to thank all of them. Most of all, I dedicate this dissertation to my wife, Hyunsuk Kim. Without her support and encouragement, this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The performance of modern many-core systems depends on the effective use of their complex cache and memory structures, and this will likely become more pronounced with the impending arrival of on-chip 3D stacked and non-volatile off-chip byte-addressable memory. Yet to date, operating systems have not treated memory as a first class schedulable resource, embracing memory heterogeneity. This dissertation presents a new system abstraction, called 'memory region', which denotes the current set of physical memory pages actively used by workloads. Using this abstraction, memory resources can be scheduled for applications to fully exploit a platform's underlying cache and memory system, thereby gaining improved performance and predictability in execution, particularly for the consolidated workloads seen in virtualized and cloud computing infrastructures. The abstraction's implementation in the Xen hypervisor involves the run-time detection of current memory regions, the scheduled mapping of those regions to caches to match performance goals, and maintaining region-to-cache mappings using per-cache page tables.

This dissertation makes the following contributions. First, its region scheduling method proposes that the location of memory blocks, rather than CPU utilization is the principal determinant for where workloads are run. It develops a new scheduling method, region scheduling, which determines the placement of memory blocks to the caches associated with workloads' processors. Second, treating memory blocks as first-class resources, new methods for efficient cache management are shown to improve application performance as well as the performance of certain operating system functions. Third, explicit memory scheduling makes it possible to disaggregate operating system functions, without the need to change OS sources and with only small markups of a target guest OS. With this method, OS functions can be mapped to specific desired platform components, an example being a file system confined to running on specific cores and using only certain memory resources designated for its use. This can improve performance for applications heavily dependent on certain OS functions, by dynamically providing those functions with the resources needed for their current use, and it can prevent performance-critical application functionality from being needlessly perturbed by OS functions used for other purposes or by other jobs. Fourth, extensions of region scheduling can also help applications deal with the heterogeneous memory resources present in future systems, including on-chip stacked DRAM and NUMA or even NVRAM memory modules. More generally, regions scheduling is shown to apply to memory structures with well-defined differences in memory access latencies.

CHAPTER I

INTRODUCTION

As heterogeneous and many-core architectures are emerging, new approaches are required to obtain performance scalability in liey of their complex memory subsystems.

On modern computer systems, memory subsystems like caches are extremely important for application performance. Contrasting existing practice in operating systems with their focus on CPU scheduling, this thesis posits that the memory subsystems in future multicore systems should be treated as first-class schedulable resources. Departing from traditional CPU-centric scheduling, we propose memory-centric scheduling methods for using the caches and other components of future memory subsystems. In doing so, we ensure careful consideration of memory subsystem characteristics like NUMAness (non-uniform memory access-ness), and we anticipate future different types of memory, such as on-chip stacked DRAM or NVRAM (non-volatile RAM), which are emerging as cost-effective alternative to current DRAM memory configurations.

1.1 New Execution Model

This thesis contributes a software approach that embraces future memory systems with a new programming and runtime abstraction, termed 'memory region'. In doing so, it contributes an execution model that moves computation to where data is located, vs. the traditional model of moving data to where the computation occurs. The model is motivated by the fact that in the mult-core era, data movements are expensive, inhibiting performance scalability. The alternative memory-centric execution model developed in our work schedules data placements and then moves computation to where data is located. Its strengths include the following. First, it can efficiently use caches because by eliminating duplicate cache lines and avoiding cache coherency traffic. This can increase the effective cache sizes seen by application. Second, it avoids data movements by moving small computation contexts vs. larger data footprints, also benefiting system power usage. Its potential weaknesses addressed in this thesis are (i) frequent migration of execution threads, requiring controls on migrations to limit consequent runtime overheads, and (ii) the need to carefully place data across a platform's memory hierarchy.

Concretely, the 'memory region' software abstraction introduced, developed, and used in this thesis may be defined as a set of physical memory pages with good spatial and temporal locality. We next outline the development and use of this abstraction in the remainder of this thesis.

1.2 Region Scheduling

Region scheduling can be viewed as a cache-aware scheduler for the CMP environment, where the last level cache is performance-critical, but even on modern processors, is not typically well utilized. This has caused hardware researchers to propose approaches like Adaptive Spill-Receive [120], NUCA (non-uniform cache architecture)[119], or Intel's smart cache [118]. Unfortunately, such approaches are limited in their ability to predict workloads' cache behavior. The region scheduling methods developed in this thesis constitute a novel, software-based approach to detecting a workload's cache footprint and then using that information to better schedule the workload.

Figure 1 depicts a simple scenario in which the two LLCs (last level caches) present on a SMP platform are not well utilized. Assume there are 4 tasks A, B, C, and D, where A and B have large working sets that fit into one LLC, while C and D have very small working sets. Existing CPU schedulers might well schedule A and B onto cache-sharing CPU cores, resulting in one LLC being over-utilized, the other being

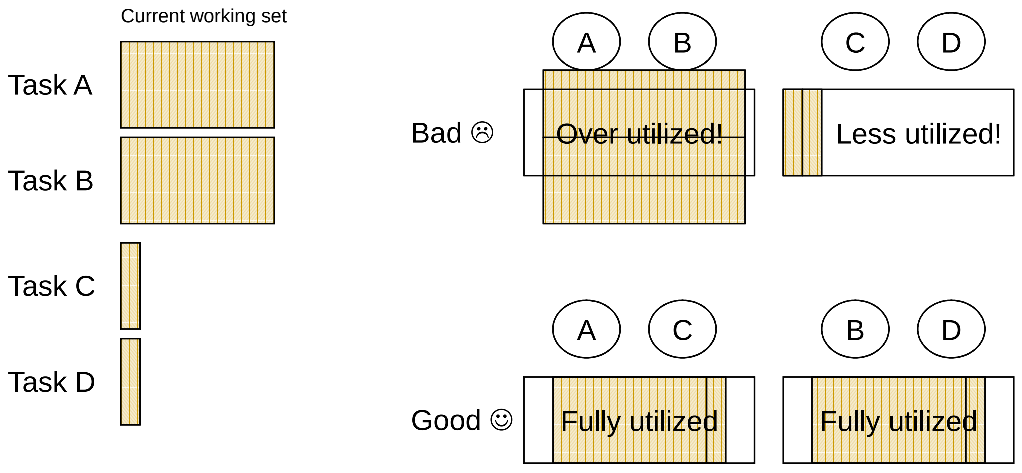


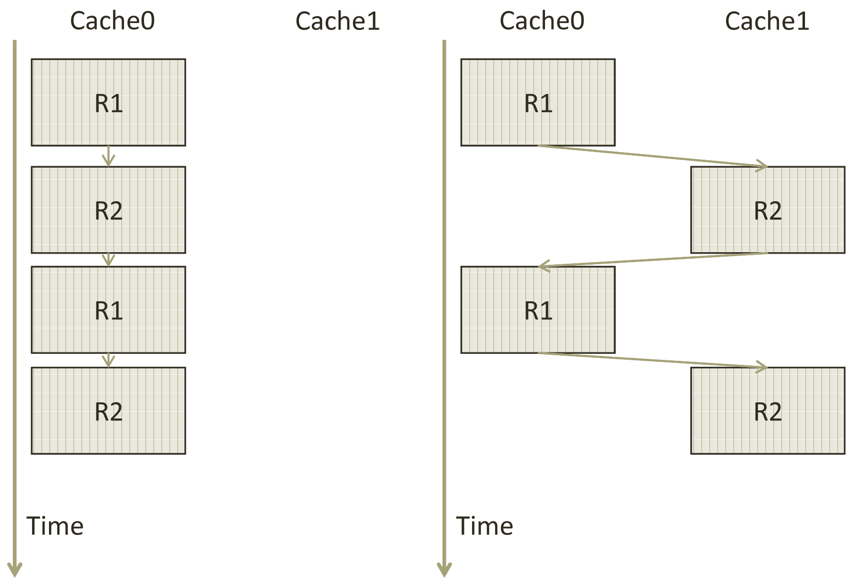
Figure 1: Unbalanced cache usage

under-utilized. This is because a scheduler that is not aware of workloads' working set sizes will schedule tasks based on CPU utilization rather than their efficient cache usage. An easy remedy for this situation, of course, is to schedule tasks A and C onto the same cache-sharing CPU cores.

The purpose of region scheduling is to schedule tasks in cache-efficient ways. To do so requires online information about workloads' working sets, to answer questions about (i) how large of these working sets, (ii) what are they comprised of, e.g., what pages do they contain, and (iii) how much content do they share?. The 'regions' identified in this work constitute such information, which is then used to appropriately schedule the tasks using them.

Region scheduling is implemented for virtualized systems using the popular Xen open source hypervisor, to realize guest-level online cache management and scheduling. Its use can substantially improve application performance and in addition, it provides hypervisor with new abilities to control guests' memory accesses.

In the remainder of this thesis, we first introduce the memory region abstraction and region scheduling. Its basic goal is to move computation to data vs. moving data to where computations are run. While this runs counter to traditional cache and memory subsystem designs that move data to where computations take place,



(a) Traditional execution (b) Region scheduling

Figure 2: VCPU view

the approach is justified by several architecture trends. First, on modern processor architectures, the expense of data movement continues to increase relative to the cost of computation. Second, data movement is subject to problems like ping-ponging and false sharing, and such problems are aggrevated by the fact that the hardware level is not informed about software decisions like work migration across different runqueues. Specifically, if software (e.g., the load balancer) moves a task that has run for a while on one cache to another cache's core, the task will experience cache misses to warm up the new cache, involving large numbers of cache line movements. Region scheduling avoids such issues by using appropriate, higher level information about tasks to reduce the effects of load balancing, to better balance the performance benefits of task migration against the costs such migrations incur. We next outline the concept of microscheduling used in the region scheduling approach, but refer the reader to the next chapter for additional detail.

Figure 2 shows how tasks are run when controlled by the region scheduler. Unlike the traditional execution model 2a, the execution under region scheduling controls

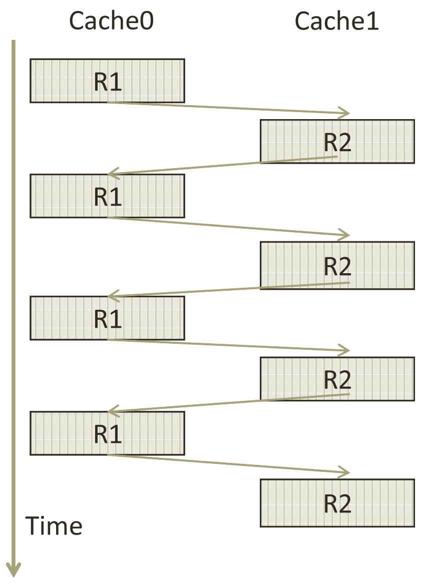


Figure 3: VCPU view

where tasks are run based on what data, i.e., which memory regions, they touch, e.g., for region switches such as R1->R2. In this context, microscheduling (usched) is defined as a switch to some region located in a different cache, initiated in response to a page touch of a page in that region. It is the page touch that triggers a page fault, due to illegal access to memory resident on a 'not-allowed' cache that triggers microscheduling. Page touch faults are handled by the hypervisor, along with appropriately setting page protection information to cause page faults as and when needed.

Figure 3 depicts the effects of frequent microscheduling. Since the overhead of microscheduling increases with short access times to a region, the effectiveness of microscheduling is governed by region access times. If those access times are too short, frequent microscheduling results in increased overhead. The challenge, therefore, is to create regions with suitably large access times but with sufficiently small granularity to capture a workload's current cache behavior. Region creation, therefore, is an inherent component of the region scheduling approach.

1.3 Disaggregated OS Services

An interesting side-effect of our ability to map certain pages to certain caches is the resulting control over the data structures contained in those pages. For operating system data structures and the services using them, this means that with region scheduling, we can directly and without the need to rewrite them, control where certain services are run. The outcome is kernel service disaggregation, to isolate some services from the effects of others and/or to protect applications from OS service activities on the cores and caches they use. Sample OS services considered in our work are file systems, networks, and scheduling, with experimental results obtained for file system and network services. For example, if the file system's inode data structures are mapped to one specific cache, file system calls are serviced only through the core group that shares that cache. We implement OS service disaggregation as a natural extension to the region scheduler, to cover all of a guest's memory, including its kernel data structures. The outcome is per guest application-level regioning coupled with kernel level service disaggregation.

In this chapter, the region framework is extended to cover the OS kernel space, and we explore not only page-to-cache mappings but also the appropriate mapping of meaningful kernel objects to pages. Object to page mappings are established via hypercalls informing the region scheduler.

1.4 Memory Heterogeneity

As new memory technologies are emerging [96], researchers have begun to study the resulting heterogeneous memory systems and their characteristics [97], with goals that include obtaining lower memory latencies and higher access bandwidths. While cost-effective, such new technologies pose challenges to operating systems regarding their effective use. This thesis explores the space of heterogeneous memory by developing a memory management scheme based on the region abstraction. Using

3D-stacked DRAM as target heterogeneous memory, with its greater bandwidth and lower latency compared to off-chip DRAM, region-based memory management aims to make best use of limited 3D DRAM capacity in conjunction with large, but slower off-chip DRAM. Specifically, the region concept is used to identify 'hot' memory pages, automatically and without user or system input, and it is those hot pages that are moved to 3D DRAM, via software-based data movement implemented in the hypervisor. Memory regions, thus, are the concept needed to identify sets of hot pages, with region movements implemented by software. Implementation of this functionality required extensions of the region scheduling framework, including hot page tracking, page movements performed in ways that are transparent to guest VMs and applications, and automation of such movements in ways that benefit application performance. An interesting side effect of this work is the ability to better manage the memory resources used by individual VMs. With current hypervisor implementations, e.g., in Xen, a single VM's memory must reside in some specific 'memory node' (there are NUMA generalizations of Xen, but those do not give explicit control to programmers [125]. Using the re-mapping methods implemented for region scheduling provides systems with the finer-grain controls needed to map VM memory to wherever appropriate.

CHAPTER II

REGION SCHEDULING

2.1 Introduction

The performance of modern many-core platforms strongly depends on the effectiveness of using their complex cache and memory structures. This indicates the need for a memory-centric approach to platform scheduling, in which it is the locations of memory blocks in caches rather than CPU idleness that determines where application processes are run. Using the term memory region to denote the current set of physical memory pages actively used by an application, this chapter presents and evaluates region-based scheduling methods for multicore platforms. This involves (i) continuously and at runtime identifying the memory regions used by executable entities, and their sizes, (ii) mapping these regions to caches to match performance goals, and (iii) maintaining region to cache mappings by ensuring that entities run on processors with direct access to the caches containing their regions. Region scheduling can implement policies that (i) offer improved performance to applications by unifying the multiple caches present on the underlying physical machine and/or by balancing cache usage to take maximum advantage of available cache space, (ii) better isolate applications from each other, particularly when their performance is strongly affected by cache availability, and also (iii) take advantage of standard scheduling and CPU-based load balancing when regioning is ineffective. This chapter describes region scheduling and its system-level implementation and evaluates its performance with micro-benchmarks and representative multi-core applications. Single applications see performance improvements of up to 15% with region scheduling, and we observe 40% latency improvements when a platform is shared by multiple applications. Superior isolation is shown to be particularly important for cache-sensitive or real-time codes.

For modern computer architectures, memory access times and caching effectiveness are key determinants of program and system performance. This is evident not only from a rich set of research on caches in computer architecture [12, 13, 14, 15, 18, 19, 20, 21, 22, 23] , but also from the wide variety of cache structures found on modern multi- and many-core platforms, ranging from single last level caches shared by from 2 (e.g., in Intels Dual-core Xeon chips) to 8 cores (e.g., in Intels Nehalem chips), to the distributed caches seen in the Larrabee chip [1].

Recognizing the importance of caching, modern methods for thread scheduling take into account cache affinity [9], avoid cache thrashing [10, 11], and/or carefully select the threads that are permitted to share a common cache [2, 3]. Leveraging such insights and in expectation of the increased importance of memory structures to the performance of future multicore platforms, our research is exploring a new approach that departs from prior process- or thread-centric scheduling methods to instead, create a memory-centric scheduler that first allocates to caches the sets of pages used by executable entities and then schedules those entities to the processors that use those caches. The schedulable sets of memory pages are termed memory regions, defined as the sets of physical pages within address spaces that currently exhibit good locality, which means that an executable entity spends significant time within each such page set region before changing its locality to reside elsewhere, i.e., in another region.

Making regions first class entities states as an explicit goal the optimization of how memory is accessed, by controlling the map-pings of regions to caches. Region-based scheduling:

- tracks the regions (and their associated physical pages) being used by each executable entity; where

- each entity can have multiple regions, but at any one time, a physical page resides in exactly one region;

- regions are mapped onto caches by system-defined mapping policies; and
- the system enforces the resulting cache-centric constraints on executable entities like processes.

This chapter presents a hypervisor-level implementation of region-based scheduling in which the VMM identifies and tracks the memory regions used in each address space, estimates working set sizes and consequent cache occupancies, and then maps regions onto caches. Mapping policies can minimize duplicate cache lines and/or cache contention or interference (e.g., to lower cache misses [32] or to improve isolation or reduce interference [26]), or they can balance cache usage across multiple processes. To attain these ends, three different scheduling policies are devised and evaluated in this chapter: (1) cache-balancing, where the system ensures high aggregate performance for the current processes running on a multi-core platform, (2) cache unification, in which the different regions used by a process are distributed across multiple caches to maximize the performance of cache-sensitive codes, and (3) cache partitioning, where software methods approximately partition the caches used by different processes to reduce interference or improve isolation. With region scheduling, it is also possible to unfairly allocate caches across different processes, perhaps to provide additional cache space to those that need it, but other than to demonstrate improvements in isolation, we do not further experiment with such techniques in this chapter.

In order to make its performance advantages available to arbi-trary applications and operating systems, region-based scheduling is implemented at hypervisor level, controlling VCPU to PCPU mappings and interacting with the hypervisors page table struc-tures (using the Xen open source hypervisor [27]). An alternative operating system-level implementation would apply region sched-uling methods to the processes and their address spaces manipu-lated by OS schedulers and memory managers (i.e., via page tables).

The outcome is a system with the following properties:

- cache-awareness the hypervisor understands the cache structure of the underlying machine, i.e., it knows which caches are associated with which P(hysical)CPUs;
- runtime region tracking low overhead runtime methods identify and track the memory regions used by the address spaces in virtual machines;
- region-based scheduling maps the V(irtual)CPUs used by a VM to PCPUs so as to match the VMs region mappings to cach-es; and performs runtime micro-scheduling, which forces a VCPU-PCPU switch to prevent the hardware from re-mapping a region when it is accessed from a PCPU associated with a different cache.

Finally, region scheduling strictly improves upon existing cache-unaware scheduling methods like those used in Unix or implemented in current hypervisors. This is because their imple-mentation reverts to unaware methods whenever regioning is deemed ineffective.

We evaluate the performance implications of region-based scheduling with representative multi-core and server applications. Experiments with the SPEC benchmark suite diagnose the potential utility and limitations of region scheduling, resulting in runtime conditions based on which we determine when region scheduling should revert to Xens standard credit-based methods. Significant performance improvements are seen for VMs running memory- and cache-intensive codes, in part by mapping their regions in ways that better leverage the combined cache sizes of multiple on-chip caches, termed cache unification. More predictable levels of performance due to improved isolation are observed for server applications with strong constraints, such as parallel codes using barriers [26] and the enterprise level VoIP codes [25] (e.g., achieving up to 40% response time improvement for the latter).

We view region-based scheduling as a first step toward design-ing schedulers that recognize the importance, if not predominance, of cache and memory structures for the performance of future multi-core applications. Complementing prior work on NUMA awareness in operating systems or hypervisors [28], region sched-uling offers system-level methods that improve and control appli-cation performance by explicitly managing their cache usage, without requiring additional hardware support [23] or inputs from applications [29]. Region scheduling can also be viewed as a first step toward systems that better support modern compiler runtimes that wish to explicitly manage the memory units places used by applications [30].

In the remainder of this chapter, Section 2.2 describes the software architecture underlying the region scheduling approach, called the region framework. Section 2.3 presents the analysis regarding regioning process. It establishes the region tracking algorithm, and working set tracking is presented in Section 2.4. Next, Section 2.5 presents performance evaluations. Section 2.6 details related work. The last Section 2.7 summarize results and future work, including speculations on potential hardware support to reduce tracking costs.

2.2 Regions

This section explains regions and the page touch methods used to implement region tracking, micro-scheduling, and the mapping of regions to caches.

2.2.1 Software Framework and Methods

A region is a set of physical pages. Regions partition memory, since at any one time; each page can belong to only one region. A region is private when its pages are accessible from only one address space, with typical private regions being those that contain heap or stack data. Shared regions, i.e., those shared among multiple address spaces, usually contain shared pages like code. Region scheduling addresses private regions, whereas shared regions are handled by standard caching hardware.

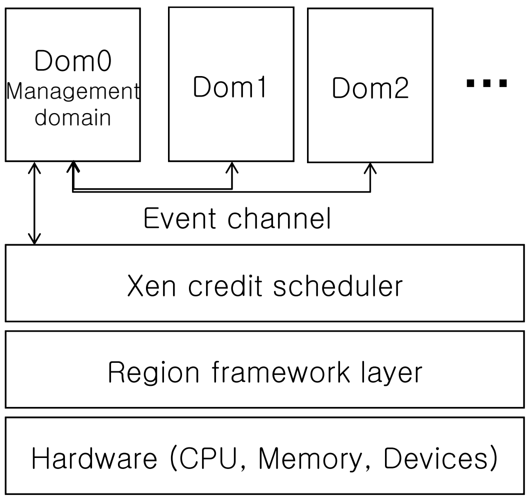


Figure 4: Xen and region framework

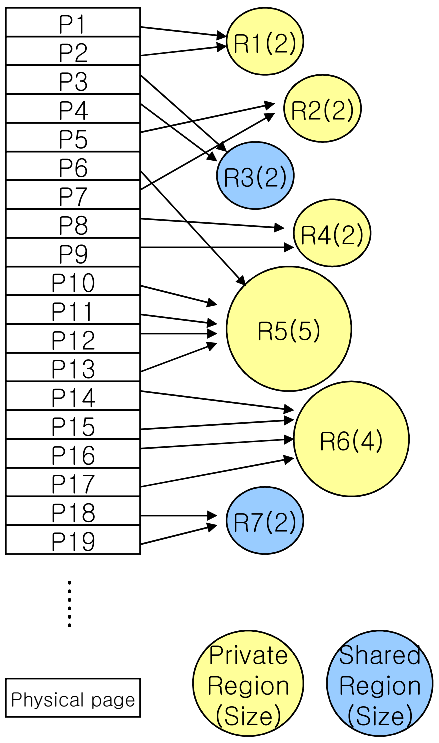


Figure 5: Physical pages and regions. (Size) is the number of pages that belong to (static) or the measured working set size at run time (dynamic).

Region-based scheduling explicitly places private regions into caches. Such mappings are maintained by having the scheduler restrict from where the regions pages can be accessed, in ac-cordance with the hardware-level association of caches with PCPUs. Access restrictions are based on specifications associated with page tables, which state, for instance, that the physical frame numbers in a region, say, 10, 11, and 12, shall be accessed only through PCPUs 0 or 2 (on our machine, both of these share access to the same cache). With such specifications, we must ensure that a region can only be accessed through the cache to which it has been mapped. This is done by raising page touch faults whenever this restriction is violated. When a fault occurs, the thread or process attempting the access is moved to one of the allowable PCPUs (i.e., 0 or 2 in this example) – termed micro-scheduling. Of course, regions may also be unmapped, and when such unmapped regions are accessed, beyond micro-scheduling, the additional option is to once again map the region to the cache used by the PCPU in question – termed opening the region.

Via page touch faults and with micro-scheduling, one can en-sure that the memory blocks in a region, e.g., pages 10, 11, and 12, exist only on cache 0, which is private to PCPUs 0 and 2. Note that this technique also minimizes the number of duplicate cache lines found in caches and in addition, it may potentially reduce cache coherency traffic and false sharing of cache lines. Further, an understanding of page to cache mappings provides approximate information about cache load, which region-based scheduling uses to better utilize the cache resources present on multicore platforms, as discussed in more detail in Section 2.4.

Figure 5 depicts a sample scenario in which the physical pages of an address space are located in different regions, private and shared ones. Each private region may be mapped to a single cache. A shared region typically exists in all caches – termed global region – an example being R7 in the figure. When there are a large number of global regions, there are fewer restrictions concerning how executable entities are run (since they can run anywhere). This means that in the extreme case of there being only global regions, the region framework layer is not active, and region scheduling reverts to standard methods, like the credit scheduler in our Xen implementation. Figure 4 illustrates this by showing how region scheduling is implemented in a layer residing between the hardware and the standard VMM scheduler.

Regions change over time, as address mappings (page table en-tries) are created or destroyed and with changes in the behaviors of the executable entities using the address space. An address spaces dynamic size its working set is the sum of the dynamic sizes of its regions, and its cache load is the sum of the mapped regions sizes. Working set size is measured at runtime (see Section 2.4). C2(11) in Figure 11 shows the working set size of C2 is measured 11, which is sum of those for R5,R6 and R7 (5,4,2 respectively in Figure 1(a)). Regions, the address spaces in which they occur, and their mappings to caches are depicted in Figure 11, which shows that regions can differ in size, that VCPU to PCPU mappings are controlled to maintain region to cache mappings, and that a single address space can be mapped across multiple caches. The latter is particularly useful for memory- and cache-intensive applications able to benefit from such cache unification.

Figure 11 also shows how region scheduling packs regions into caches, where the two address spaces A1 and A2 run on cache C1, while A3 runs on C2 because its working set is larger. Cache load is shown in parentheses, the cache with a lower load being considered emptier when regions are bin-packed into caches.

2.2.2 Life Cycle of a Memory Region

Figure 6 to 10 shows the life cycle of a memory region, using an example. When a new virtual-to-physical mapping to a physical page is created, a region is born. In Figure 6, R1 is born with first page table entry (mapping, the first arrow) and it increases its size (the number of physical pages it manages) to 2 with the second arrow. Now

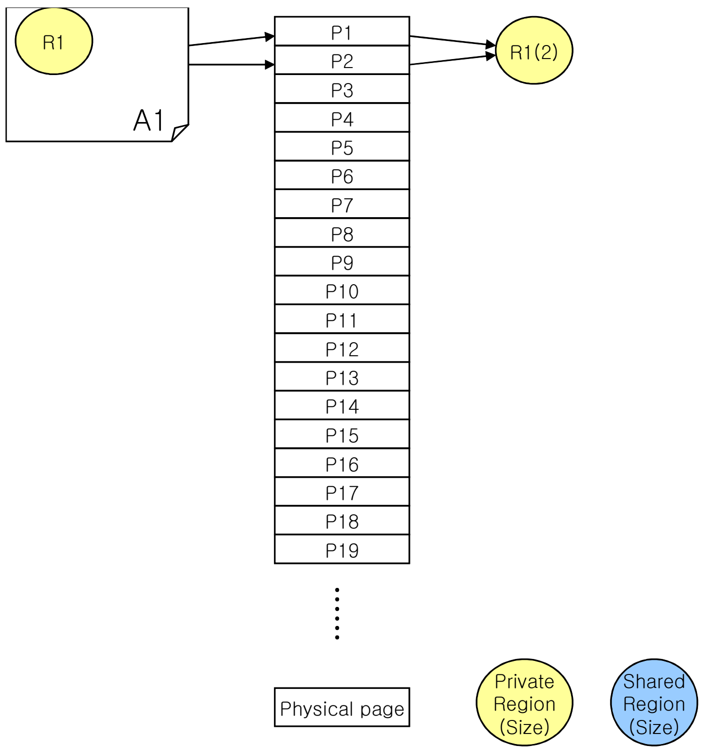


Figure 6: Life of region – born

assume that this process (or address space) creates more mappings in page table by requesting more physical pages, then we see more arrows from A1 as in Figure 7. New physical pages may be combined to existing region, or can create new region. Initially it can be merged to the existing one and such regions are called 'seed' region. It just plays role of a initial pool of pages. Later it may be split into regular regions once it goes through some learning phase. For now, we assume that we have two new regions R2 and R3 somehow in this example. Therefore, the address space A1 now has three private regions R1, R2, R3. Now if another process creates mappings to two pages P3, P4, then R3 becomes a shared region between two processes A1 and A2. To be exact, the first mapping would split R3 into new shared region R3' and existing R3 and second mapping would create new shared region R3", then R3' and R3" get merged which effectively turns the private region R3 into the shared region R3.

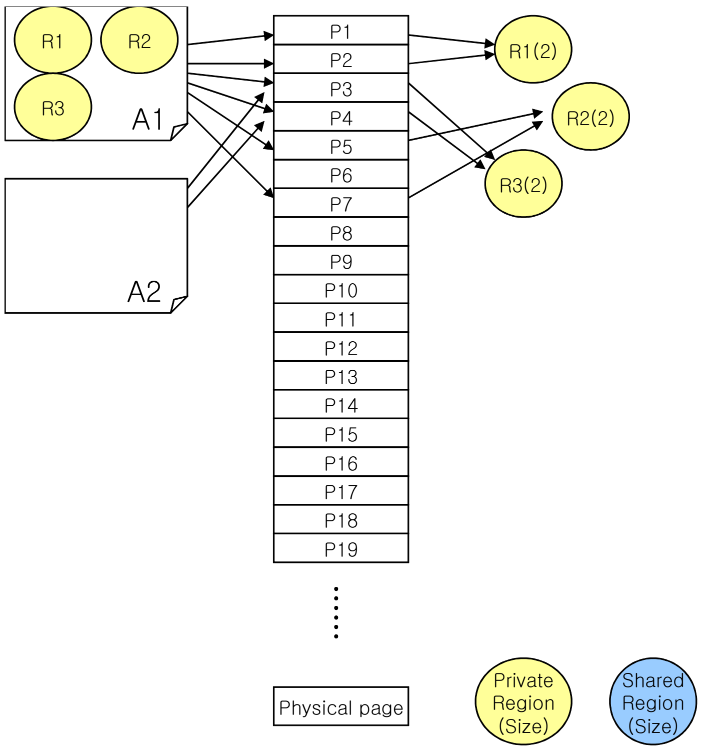


Figure 7: Life of region – new process

In Figure 8, A2 creates more mappings and it results more private regions R4, R7. Similarly A3 may come into picture and create more mappings as in Figure 9. This process, A3, has bigger working set size and naturally the private regions are bigger in its sizes. If A1 terminates and removes its mappings to pages as in Figure 10, their private regions dies accordingly and it may potentially turn a shared region into a private region.

This example not only shows the life of a region, but it also indicates different region types. The seed region plays the role of a pool of pages, and it splits into regular regions later. Sometimes, such regions may be merged to form bigger region. That is, regions can dynamically change as per the guest's behavior.

2.2.3 Region Scheduling Implementation

Table 1 describes the data structure maintained for each region

- pgd: if a region is private, it belongs to a certain address space and pgd points

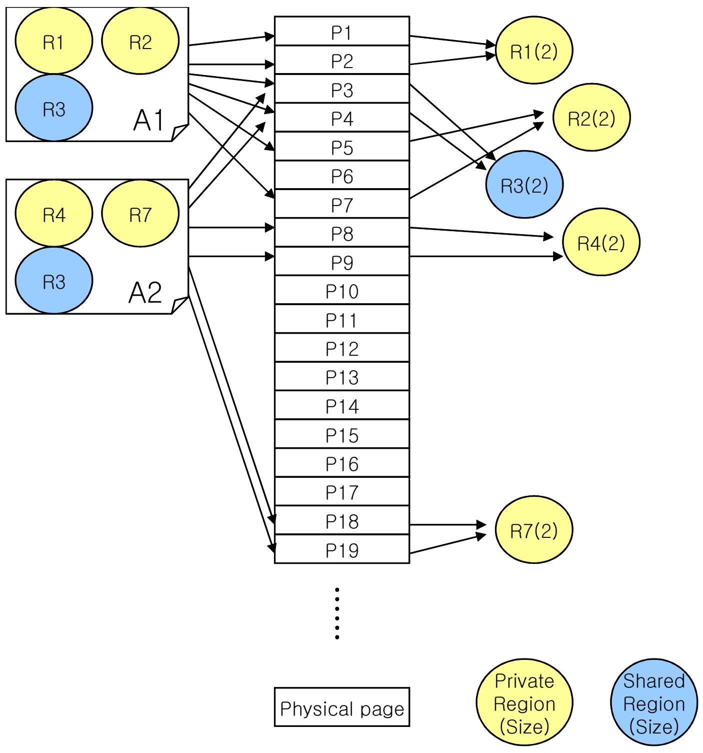


Figure 8: Life of region – shared region

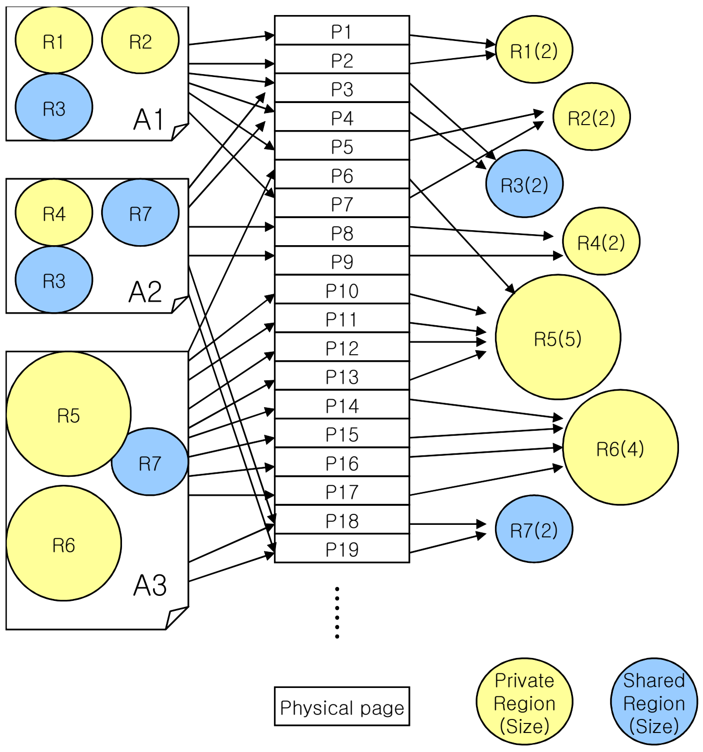


Figure 9: Life of region – another process with bigger working set

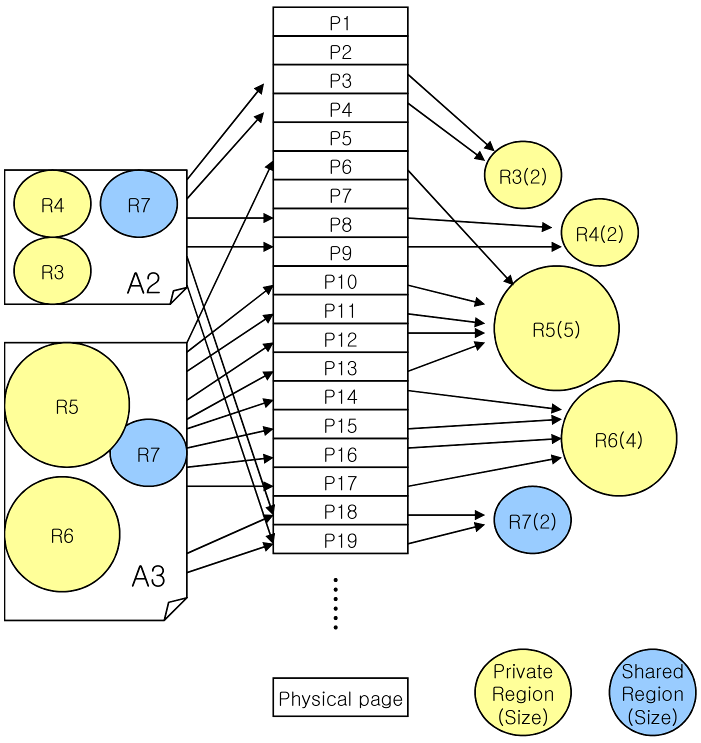


Figure 10: Life of region – death

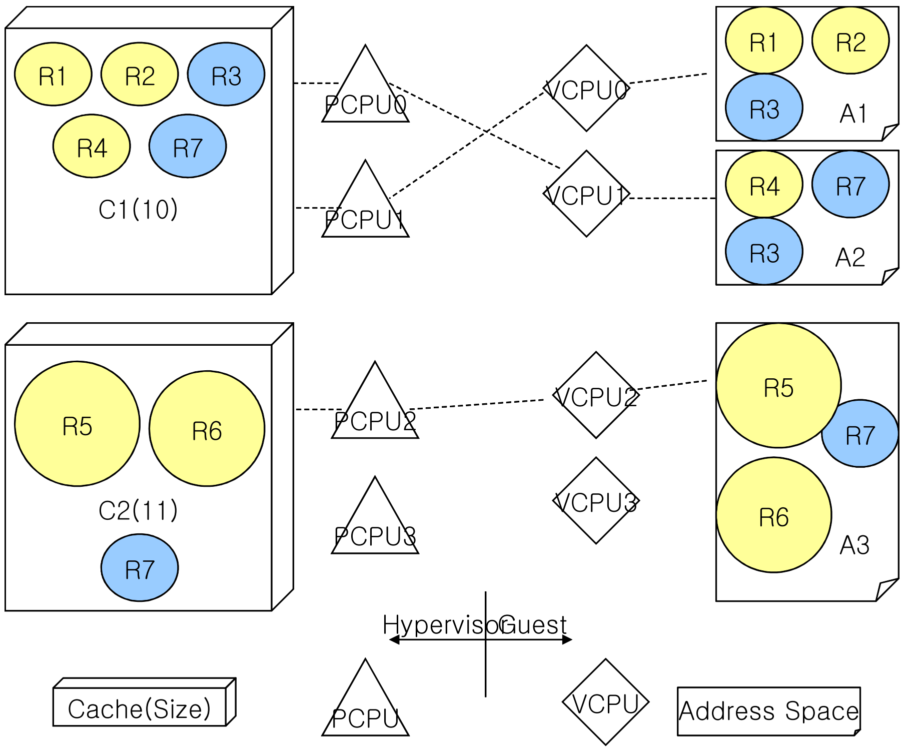


Figure 11: How regions interact with caches and address spaces

```
struct region_t {
        struct page_dir *pgd;  // address space if private region
        atomic_t vr_refcnt;  // reference count
        struct list_head list[MAX_CACHE];  // mapping to caches
        spinlock_t lock;  // lock
        struct list_head rmaps_list;  // reverse maps
        unsigned short int frame_count;  // static size
        unsigned short int rmap_count;  // # of reverse map
        unsigned short int flags;  // flags
        unsigned short abit[MAX_CACHE][32];   // histogram
};
```

Table 1: System-level representation of regions

to the address space.

- refcnt: a region has a reference count, which is used to deallocate a region when it is no longer used.
- list: a region is mapped to some number of caches (typically, to only one)
- lock: a region structure is protected by basic locking primitives
- rmaps_list : a region has a reverse map to the page table so that region to page mappings are easily changed
- frame_count : the number of pages a region manages.
- rmap_count : the number of reverse map to the page table for optimization
- flags : flags that indicates current state of a region
- abit : tracks the access bits

The reverse map is important because when a region is mapped to a cache, all page table entries to all pages in the region must be modified in order to ensure that only those address spaces running on the right cores are permitted to access it. It is easy to maintain because Xen must already intercept all page table modifications.

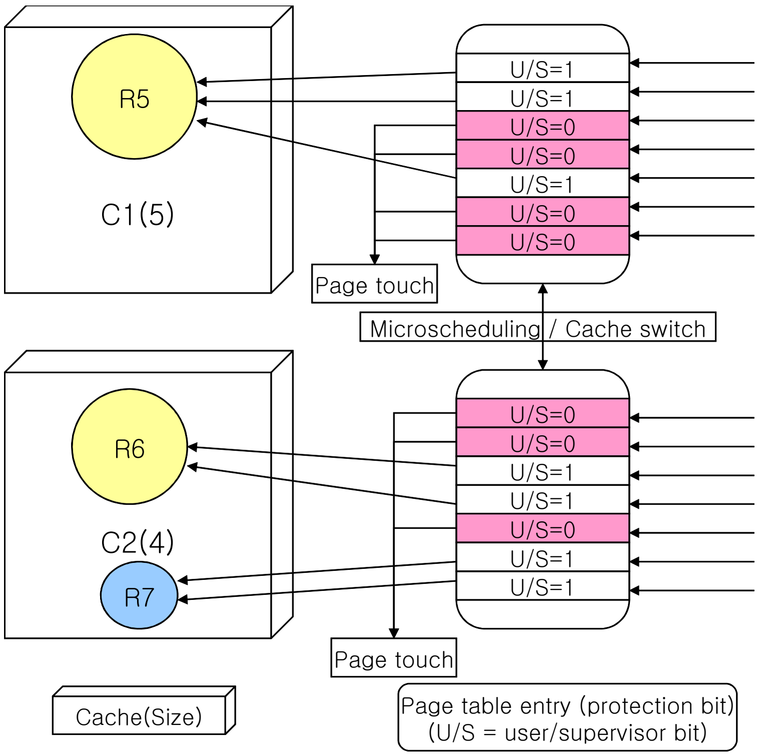


Figure 12: Page touch and cache switch

For other address spaces, we simply clear the protection bit in the page table, thereby causing an access fault when any of them attempts to use the page. Such page touches are not propagated to guests, but are transparently handled by the region scheduling framework. Figure 12 depicts this. Note that without a reverse map, these actions would require an expensive complete page table scan.

2.2.3.1 Page touch and cache switch

As indicated in Section 2.2, upon page touch, the region scheduler has two options: (i) to allow the access, which requires mapping the touched region to the current cache, termed opening the region; or (ii) to move the executable entity to the CPU whose cache is currently allocated to the region, termed micro-scheduling. An entity is micro-scheduled in order to force it to run on a different cache. For such a cache switch, we inspect all of the regions used by that entity, set the protection bits for the mapped regions to the target cache opening the regions and clear it for the

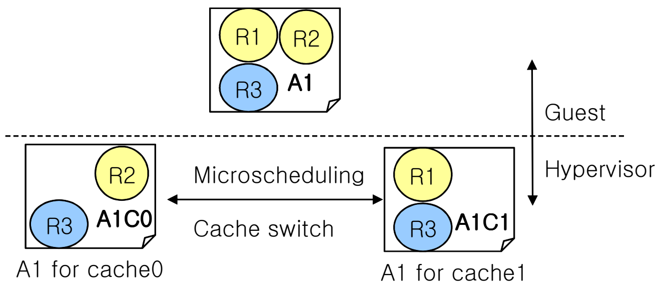


Figure 13: Per-cache page table

other regions closing them. Figure 12 shows how page tables are manipulated for each cache switch. Global regions, which are already mapped into both caches, are skipped since there is no need to manipulate them.

Page table manipulations are also used to collect information about an applications behaviour in terms of memory accesses and to track its working set. By simply closing a region, one can detect when a VCPU enters it, for instance, which we use to help assess working set size. By ‘closing regions that have not been accessed for a while, region management is optimized in terms of the number of open regions it must consider.

2.2.3.2 Micro-scheduling and cache switches

As evident from the description above, micro-scheduling involves cache switching. This could be expensive if it required the hyper-visor to explicitly touch all of the address spaces private regions and their page table entries. We eliminate this over-head by main-taining per-cache page tables. This is shown in Figure 13, where the hypervisors page tables A1C0 and A1C1 jointly have the same contents as the guests cache table A1; they differ only in the protection bits used to ensure that regions are open or closed with respect to certain caches. This also enables multiple threads in a process to run across caches.

Beyond cache switching, the other costs of micro-scheduling con-cern VCPU/PCPU re-mappings. Figure 14 depicts a case in which one VCPU runs on four PCPUs, where

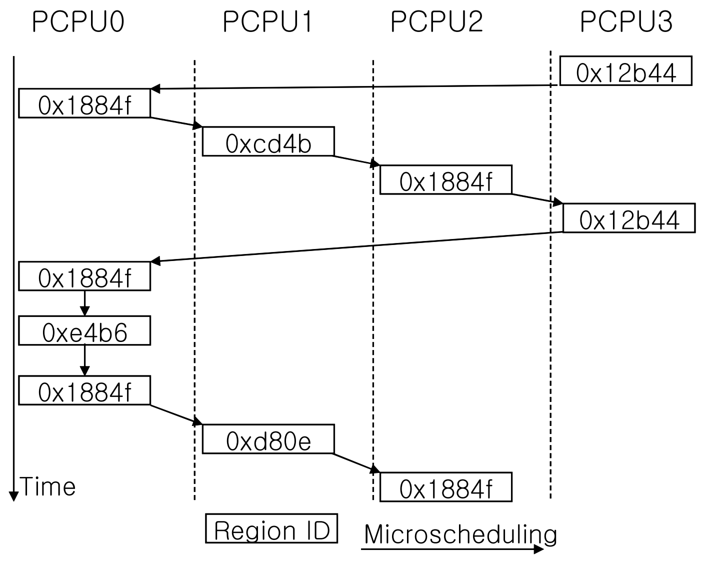


Figure 14: Microscheduling with 1VCPU

the hexadecimal in each rectangle is the unique ID for each region used by the VCPU. In this hardware configuration, Cache 0 is shared by (or local to) PCPUs 0, 2, and Cache 1 is shared by (local to) PCPUs 1, 3. Cache 0 is allocated to Regions 0x1884f, 0xe4b6, and Cache 1 is allocated to Regions 0xcd4b, 0xd80e, and 0x12b44. For example, since Region 0x1884f is mapped only to Cache 0, when the VCPU tries to access this region, it is scheduled onto PCPU0 or PCPU2. Note that it is the standard scheduler (such as Xens credit scheduler) that determines which PCPUs are allocated to them. Micro-scheduling, then, simply makes sure that VCPUs always run on those PCPUs that are associated with the caches allocated to the regions they are currently accessing. Potential performance opportunities and liabilities derived from these constraints are discussed next.

Figure 15 depicts a more complex case in which 4 VCPUs run on 4 PCPUs, where VCPUs access some regions only through PCPUs 0, 2 and others through PCPUs 1, 3. For example, the region 0x13d83 is mapped to Cache0, and 0x1225e is mapped to Cache1 (see Figure 16 for the associated region-to-cache mapping). We can see how the region framework balances cache loads from these figures. We discuss cache

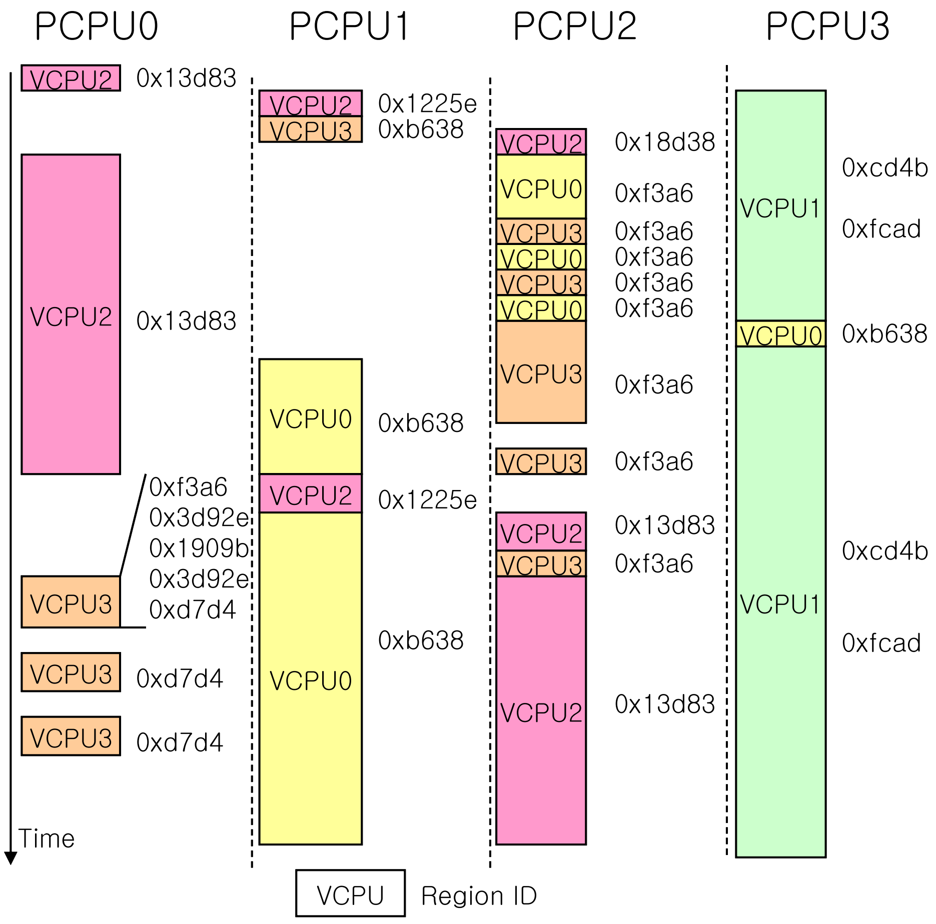


Figure 15: Microscheduling with 4 VCPUs

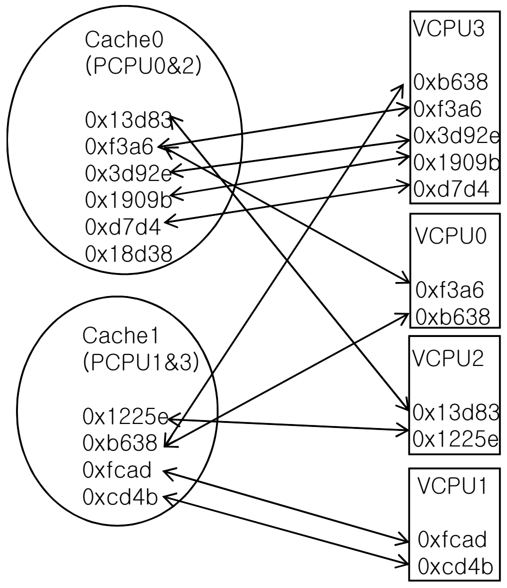


Figure 16: Mappings among regions, caches, and VCPUs

balancing in Section 2.4.

A potential side effect of cache balancing is that PCPUs may experience additional idle time. This is illustrated by the idle time observed on PCPU1 in Figure 15, which occurs because VCPUs 0, 3, 2 are running on Cache 0 (PCPU0, 2) while only VCPU1 is running on Cache 1 (PCPU1, 3). In fact, VCPUs 0, 3 are compet-ing for PCPU2, while PCPU1 is idle. This transient imbalance of VCPUs on two caches is due to restrictive region-to-cache map-pings. Such an imbalance is desirable if VCPU1 greatly benefits from its exclusive access to its cache (e.g., for cache-intensive codes), but at the same time, it may increase the latencies experi-enced by other VMs due to the effectively smaller cache sizes made available to them. The conflict is mitigated (1) when there are more VCPUs (due to VM-internal parallelism or consolidated VMs), so that it is likely that other VCPUs can be found to fill this gap, or (2) when there are more PCPUs per cache. Further, we use an additional method to prevent CPU idleness, in which instead of micro-scheduling VCPUs, we manage regions in order to handle this conflict between CPU and cache workload balancing. Results on such cache balancing appear in Section 2.5.3. Our final solution is to simply permit the region framework to make regions global (region opening) to prevent CPU idleness. Such degeneration to standard scheduling is useful for codes that do not depend much in performance on efficient cache use.

2.3 Regioning

This section explains region identification and tracking. At two extreme ends, all (private) physical pages in an address space could be placed (1) into a single region (too coarse-grained) or (2) into many single-page regions (too fine-grained). The first says that only entire address spaces can be mapped onto caches, whereas the second states that we have little information regarding its locality. To determine page-to-region associations, therefore, requires runtime methods that analyze the benefits and

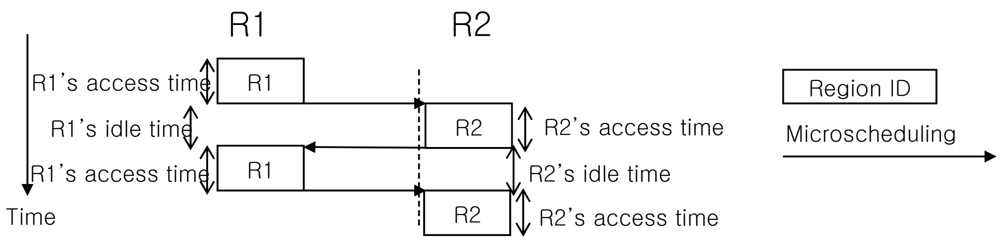


Figure 17: Two region memlats access and idle times

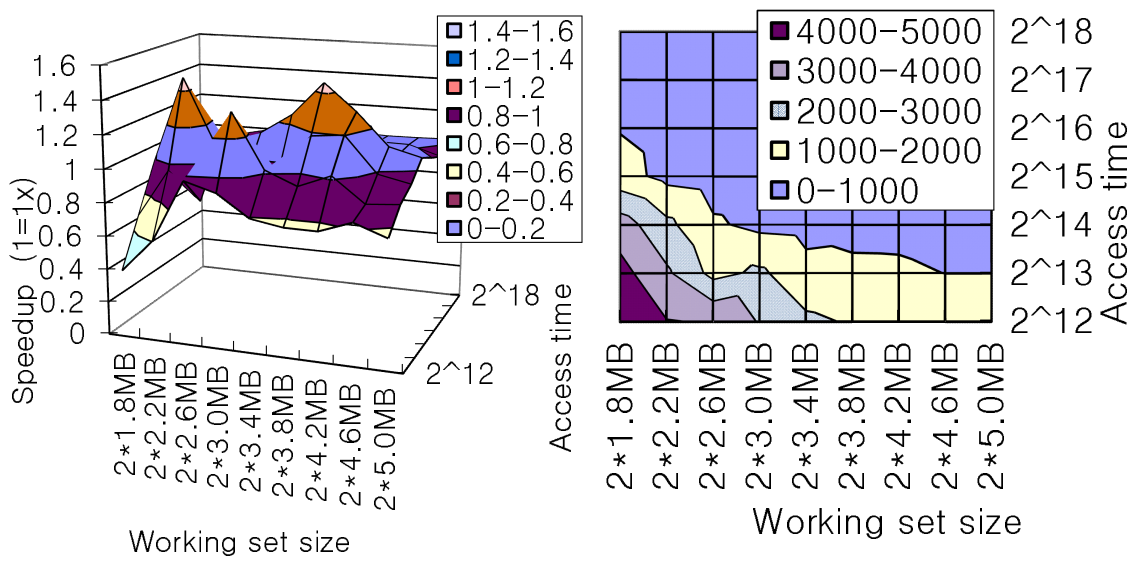
overheads of region formation and management, and of the micro-scheduling actions necessary to enforce region to cache mappings. This section identifies such regioning conditions and uses them to construct regioning algorithm.

2.3.1 Cache Unification

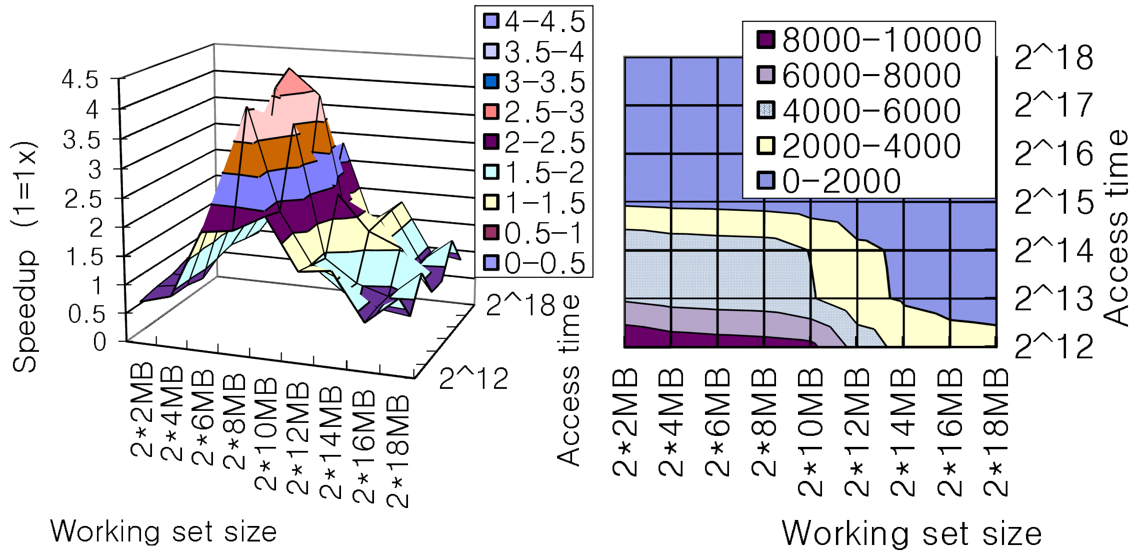
A simple single-threaded micro-benchmark, termed memlat (memory latency) based on [4], is used to assess the potential utility of cache unification. This memlat has two identically sized regions, which it traverses randomly for some given number of memory references and across a given number of pages, termed region access time, before its execution switches to the other region, which results in a consequent value of region idle time (see Figure 17).

In the experiment, instead of confining the memlats regions and thus, its execution to one cache, we map its two regions to two different caches and micro-schedule it across the associated PCPUs, then compare it to the cache confining case. This is done for two different generations of machines (Clovertown and Westmere – see Section 2.5 for additional detail)

Figure 18 shows the normalized performance of the two region memlat when caches are unified, where values greater than 1 denote improved performance compared to the case of cache confinement. The x axis is working set sizes (2*2MB means two 2MB regions), and the y axis is access time in the number of elements touched before a region switch occurs. In this section, access time is expressed in memory access



(a) Clovertown



(b) Westmere

Figure 18: Two-region memlat across caches. Normalized performance (3D) and microscheduling rate per second (contour view)

count rather than actual time. The figure shows that improved performance appears in the center, not at the edges of the graphs.

There are several interesting insights from these simple experiments. First, substantial opportunities exist for gaining perfor-mance improvements from using cache unification, up to 45% for Clovertown and over 300% for Westmere. This is despite signifi-cant numbers of micro-scheduling actions in Figure 18, with rates ranging from 295 to 2240 per second for successful cache-unification near the center for Clovertown, and with rates ranging from 180 to 5100 per second for Westmere. Second, Westmere has a greater range in which improvements are seen (>1), and this is because of its relatively lower cost of micro-scheduling (see Section 2.5). This reflects the fact that computer architects have taken great pains to reduce the overheads of context switching on modern CPUs. Also, third, we can see that the microscheduling rate in-creases as region size and access time decrease.

Second, on Clovertown, performance improvements are mar-ginal when access times are high (>= 2ˆ16) because in those cases, there are relatively few micro-scheduling actions that permit applications to benefit from cache unification. Third, as expected, when access times are too short (<=2ˆ12), the large number of micro-scheduling actions create overheads that outweigh the utility of cache unification. The outcome is Condition 1, which states that access time must be in some platform-specific range (i.e., these normative experiments have to be performed for each platform used) in order for region-based scheduling to benefit from cache unification.

Conditions 2 and 3 concern cache working set sizes (recall that a cache working set is defined as the sum of the sizes of all of the regions being used). First, there are negative effects when working sets are too small (<=2*1.8MB for Clovertown, <=2*4MB for Westmere), because placing working sets that would fit into a single cache into two different caches simply causes the added overheads of micro-scheduling. Second, when a working set is so large that it does not fit into the unified two caches,

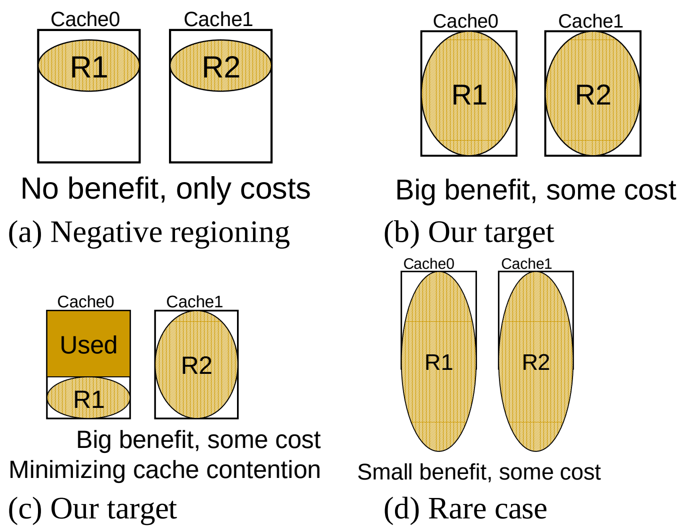


Figure 19: Cache working sets and cache sizes

then again, there are no benefits from region scheduling, since there would be cache misses both with region scheduling (and the additional overheads associated with it) and without region scheduling. Condition 2, then, states that the total working set size must be larger than that of a single cache, and Condition 3 states that each working set must fit into the unified space provided by both caches. Conditions 1-3 are shown pictorially in Figure 19. As stated earlier, actual benefits and costs vary across platforms, but from the Westmere vs. Clovertown results shown here, it appears that future platforms will likely further tilt the playing field toward our more explicit methods for cache management.

2.3.2 Bottom-up Regioning

We are now ready to explain how regioning is performed. Re-gions are captured at runtime. There are two extremes: (1) random regioning where physical pages are placed into regions randomly, which would cause high rates of micro-scheduling, and (2) single regioning, where all pages are placed into a single region, thereby effectively disabling region-based scheduling and entirely avoiding micro-scheduling

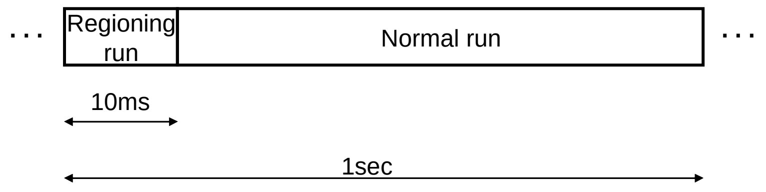


Figure 20: Regioning run (p = 1%)

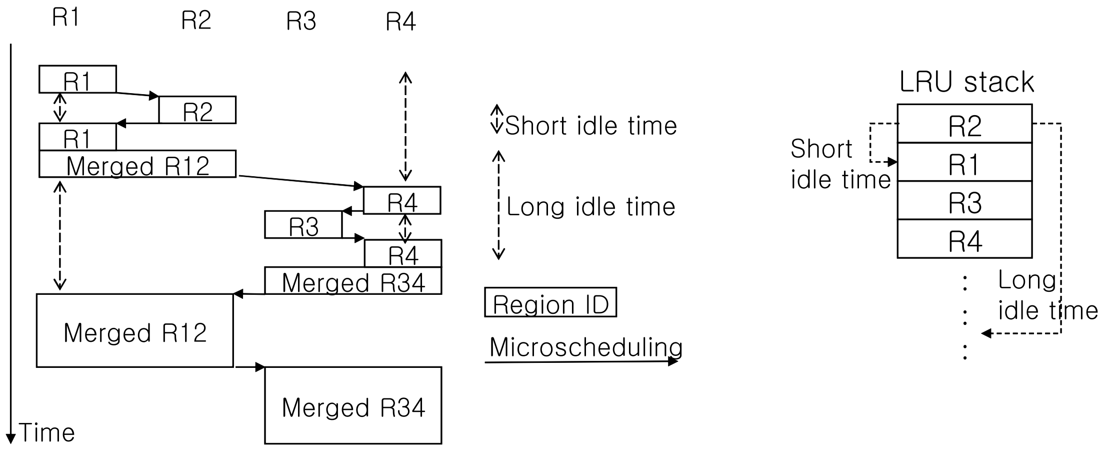


Figure 21: Regions and region merging via LRU stack

overheads. Between these two extremes, we use Conditions 1-3 formulated above to assess the utility of regioning, and we identify and track regions using a sampling-based clustering technique, a bottom-up approach based on the notions of access and idle times.

Each address space runs for 1% of its time in regioning mode (see Figure 20), in which initially, there are only single-page regions that are then repeatedly merged to form suitably sized regions to contain application locality. In addition, at the end of each regioning phase, some regions are torn down in order to prevent them from becoming too large and/or to capture substantial changes in application behavior (e.g., phase changes). Finally, for accuracy, regioning performed across interrupt handlers and system timers is adjusted to correctly consider such system activities.

Figure 21 depicts a scenario in which multiple smaller regions with longer vs. shorter inter-region idle times are merged into a smaller number of larger regions. This is done as follows. First, during the regioning phase, all region switches are detected because initially, all regions are closed (except for global regions). This means that entering a region causes a page touch that is visible to the system. This makes it possible to construct a stack of regions based on the (prev_region =>next_region) occurrences. Second, when a new region is entered, the previous region is closed, so that only one region (the current region) is open at any given time. This makes it easy to measure the access and idle times for all regions.

As Figure 21b shows, the idle times correspond to the LRU distance between regions. Therefore, a long idle time indicates a locality change, whereas a short idle time between two regions is a strong indicator for merg-ing them, both of which are shown in Figure 21. Using a threshold q to determine short idle times based on the memlat measurements explained earlier, we merge regions when idle time is less than q and take no action otherwise. Note that a low threshold results in fine (small) regions, while a high threshold creates coarse (bigger) regions.

All regions are opened to resume normal execution after the regioning phase has completed. This entering/exiting of the re-gioning phase could be expensive, however, because all regions in the address space should be closed/opened when this occurs. This is optimized by introducing per-mode page tables in ways similar to what is discussed in Section 2.2. As a result, the regioning phase can be entered by simply switching to separate regioning-phase page table that already has closed entries.

2.3.3 Region Types

There are several types of regions. Initially, all regions are seed regions. When locality is captured in the regioning phase, they become regular regions and once mapped to some cache, they are termed local regions. As stated earlier, there are also global regions not subject to region scheduling.

Differentiating global from other regions is done as follows. At each page-touch

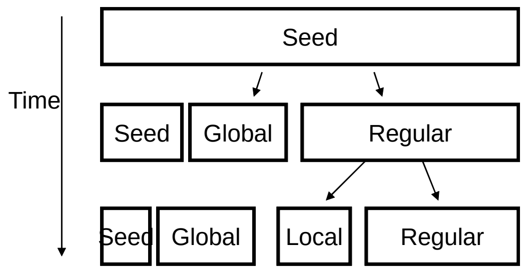


Figure 22: Types of regions

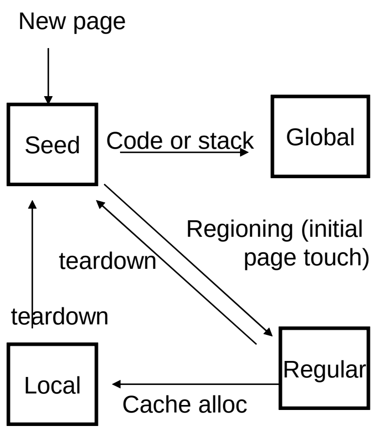


Figure 23: Region state transitions

```
(1) Start regioning (close all regions except global regions)
(2) Regioning phase (merging)
(3) End regioning (open all regions)
(4) Cache balancer does cache allocation, cache balancing.
(5) Teardown (pick largest local region from regioning phase and
    make it into a seed region.)
```

Table 2: Regioning process

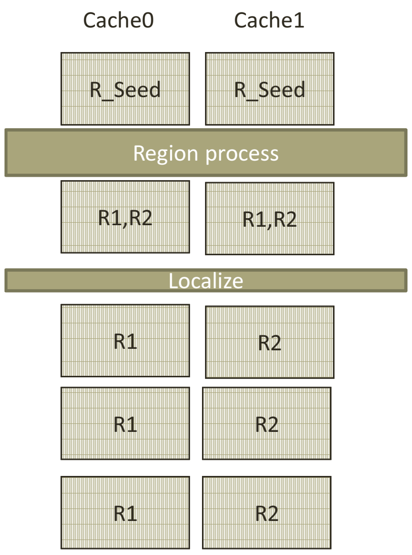


Figure 24: System view

from a seed region, the new page is determined as code if the faulting address equals the eip (program counter) register. If the faulting address is near the stack pointer, it is de-termined to be a stack page. Both code and stack pages are classified into global regions, and thus, do not further participate in the regioning process. The type transitions shown in Figure 23 apply to all other pages. In summary, the process in Table 2 is used for regioning. Figure 24 depicts region state transitions over time.

We have not yet explained when large regions are destroyed (by turning them into seed regions). This is done in conjunction with cache allocation to regions. Specifically, cache allocation is performed after regioning is completed, by picking a regular region and making it into a local region (i.e., mapping the region to some cache). Next, we select the largest local region determined in the regioning phase (which is marked in that phase, so this is an O(1) operation), and denote it as a seed region, thereby initiating the process of tearing it down. We never consecutively tear down the same region, however. In this fashion, we incrementally build (and tear down) regions in response to observed program behavior. Finally, a region may shrink during a run when an abnormally high microscheduling rate is detected, by

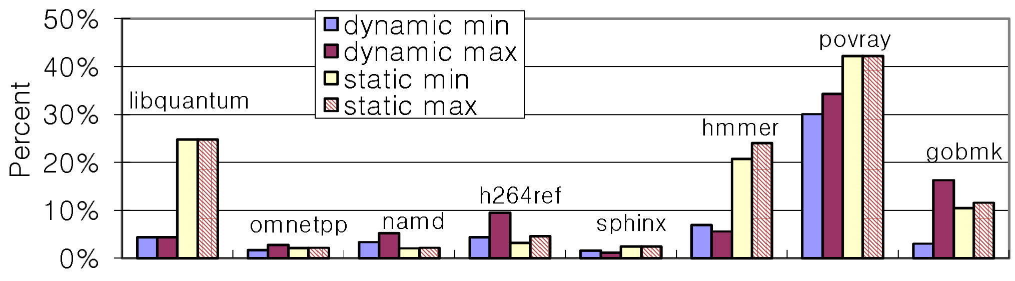


Figure 25: Portion of shared pages (observed min/max value)

excluding from the region the page that causes it.

Regioning is independently conducted for each core, the current policy doing it at every 1 second of CPU time for each address space. Thus, long-lived processes will experience more regioning actions, whereas short-lived ones may not experience any regioning at all (i.e., if they live less than 1sec). For multithreaded applications sharing an address space, each of the different threads (i.e., the cores on which they run) enter the regioning phase at a different time, thereby avoiding concurrent use of the shared page table.

2.3.4 Global Regions

Inappropriate placement of shared pages and stack pages can cause unnecessary micro-scheduling overheads. An example is to map the glibc code onto only one cache, which would cause virtually all processes to frequently micro-schedule to glibcs cache. To address this issue, we declare all code pages (and similarly, the stack pages) to be global, which causes some level of cache line duplication. The resulting overheads in terms of cache space usage are moderate, however, as shown by the measurements in Figure 25 assessing the portion of shared pages in the SPEC benchmark suite. The figure shows the percentage of observed shared pages at runtime, both actually accessed (dynamic) percentage and the static percentage seen in page tables.

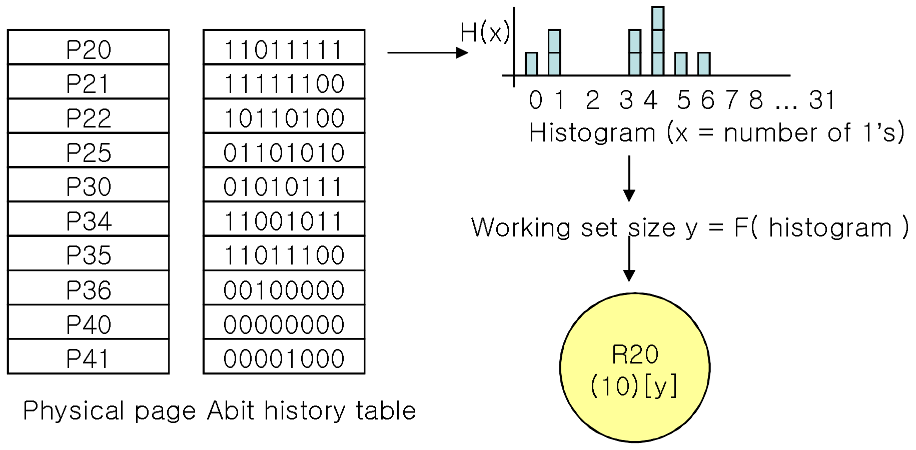


Figure 26: Access bit history and regions working set size

2.4 Working Set Tracking

Working set sizes are determined dynamically, using the access bits (A-bits) in page table entries. Specifically, every 100ms (as virtual time for each address space), the page table is scanned, and the access-bit history is recorded in a 32bit word. Only currently open regions are scanned to minimize overhead. For the example shown in Figure 12, if it is running on C2, only the part of the page table corresponding to R6 and R7 would be scanned, for instance. The access bits gathered over time form an access bit history (i.e., 3.2 seconds worth of access history) for each page. This is also termed the pages access pattern recorded as region histograms based on their pages access histories.

Figure 26 shows region R20s details. From its 10 pages ac-cess histories, it builds a histogram by counting the number of 1s, and it calculates its working set size y using a heuristic moving average function F below that takes the histogram as its input. The weights are determined experimentally.

$$y = \sum_{i=26}^{31} H(i) + \sum_{i=20}^{25} \frac{3}{4} H(i) + \sum_{i=14}^{19} \frac{1}{2} H(i) + \sum_{i=8}^{13} \frac{1}{4} H(i) \qquad (1)$$

Consistent with this function, we define cache load as the sum of the dynamic sizes of mapped regions. Interpreting this value as cache occupancy, the cache balancer

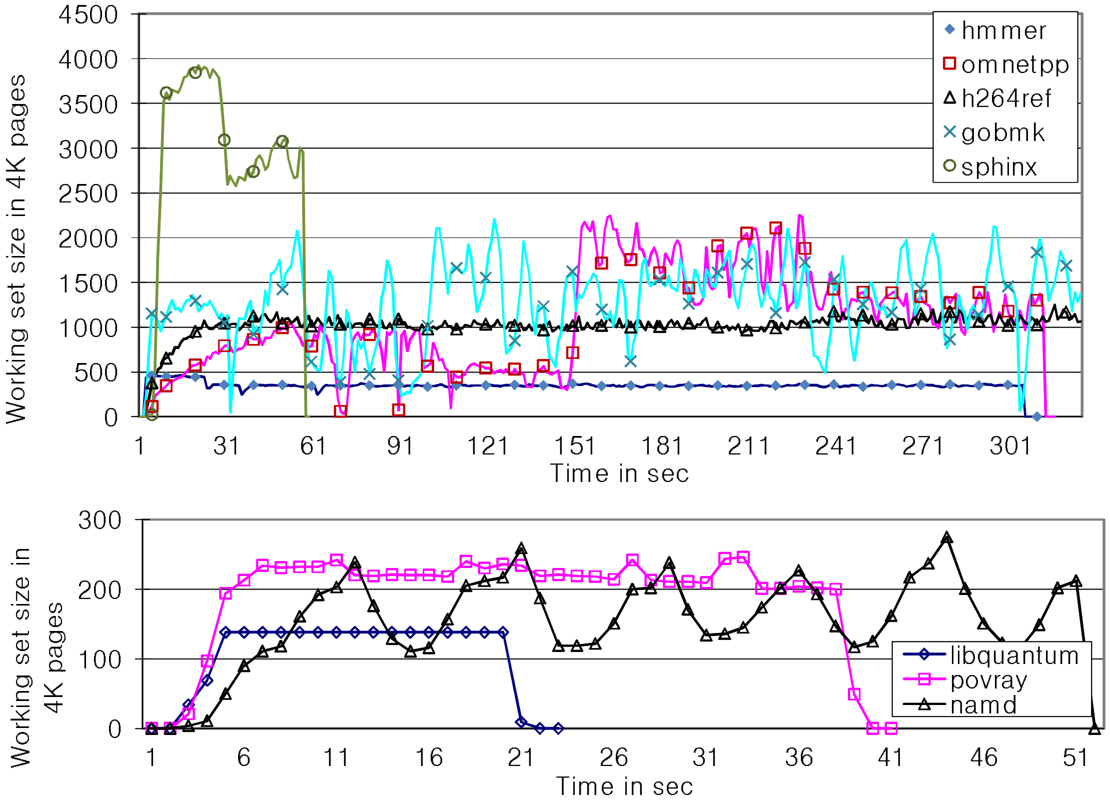


Figure 27: Tracking working set size

uses it to determine cache imbalance; a simple greedy algorithm periodically balances cache usage in conjunction with the regioning process.

Since access bits are gathered every 100ms, and a 32bit word is used to store its history, roughly the past 3.2 seconds are reflected in the working set sizes used for region scheduling. Figure 27 shows the evolution of cache working set sizes observed over time for select benchmark codes, which the cache balancer would use.

2.5 Experimental Evaluation

Region scheduling is evaluated on two generations of Intel plat-forms. The first, labelled Clovertown in all figures below, is an older machine with Intel quad-core Xeon X5365@3.00GHz cores with 1GB RAM. The caches are an 8-way L1 cache (32K Da-ta+32K Instruction) and a 16-way 2x4MB L2 cache. The cache line size is 64bytes. The second, labelled Westmere in all figures below is a newer machine with

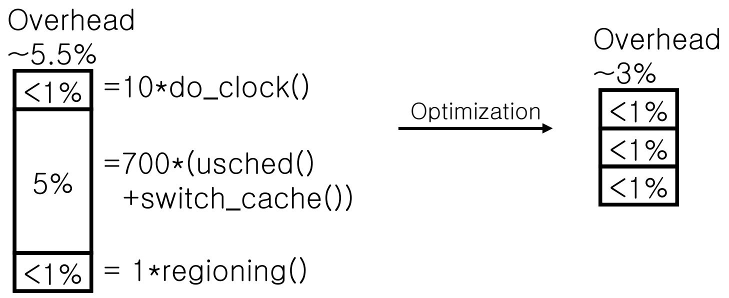


Figure 28: Overhead

two Intel six-core Xeon X5660@2.80GHz sockets with 12GB RAM. The caches are a 4-way 32KB L1I, 8-way 32KB L1D, 8-way 256KB L2, and shared 16-way 12288KB L3.

2.5.1 Overhead

Regions are enforced by the hardware MMU. Once page protection bits are set during opening/closing regions, no additional runtime overheads are incurred during execution for enforcing regions. Full TLB flushes are avoided by using optimized instructions like invlpg.

Four major overhead sources are the do_clock(), usched(), switch_cache(), and regioning() calls, which perform page table scanning, microscheduling, page table updates, and regioning respectively. Using the two-region memlat with a small working set size, we conduct an extreme case experiment to assess these overheads, by choosing working sets that perfectly fit into the two different machines caches, thereby eliminating all potential benefits derived from micro-scheduling. When not using the per-cache page table optimization, total overhead is measured to be roughly 5.5% with 700 microscheduling per second. Performing the do_clock() call ten times and making one regioning() call results in less than 1% overhead, but the switch_cache() call constitutes over 90% of total overhead, which is effectively eliminated using said optimization. This results in constant-time micro-scheduling, its composite time comprised of page fault, context switch, runqueue manipulation, IPI, and TLB flush.

Figure 28 briefly shows this. Micro-scheduling is measured as 47600 cycles (i.e., 15.86 us, 1.5% for 1000 uschedule actions) for Clovertown, and 17800 cycles (i.e. 6.357 us, 0.6% for 1000 uschedule) for Westmere. Do_clock() has some overheads de-pending on page table size, but it is several milliseconds in most cases (less than 1%). Regioning overheads benefit from the optimization that uses per-mode page tables, where an upper bound on these costs is defined by the sampling rate p. Naturally, over-heads are even lower, close to two TLB flushes, for codes that operate with stable regions, like libquantum. That overall overhead is measured to be less than 3%, typically 2% on both machines.

2.5.2 Microbenchmarks

To reduce cache contention, the cache balancer dynamically re-maps regions based on cache loads. For example, on machine Clovertown, when running two processes of 4MB working set size and two processes of 16KB working set size, with region scheduling, the cache balancer ensures that the cache is shared by the pair of 4MB+16KB processes. This improves performance by more than 50% for all processes compared to a region unaware mapping in which two 4MB processes share a cache. Similar results are observed on machine Westmere, using 12MB and 512KB working set sizes, respectively.

Figure 29 shows simple experiments on both machines, in which we run two 4MB memlats + 16KB memlat + a SPEC benchmark on the machine Clovertown, and two 12MB memlats + 512KB memlat + a SPEC benchmark on the machine Westmere. Depending on scheduling, the SPEC benchmark experiences different levels of cache contention. The cache balancer improves performance by correctly pairing processes onto caches and miti-gating cache contention. The figure shows that there is substantial potential for performance improvement for all SPEC benchmarks. Or stated negatively, without cache balancing, SPEC programs experience significant levels of

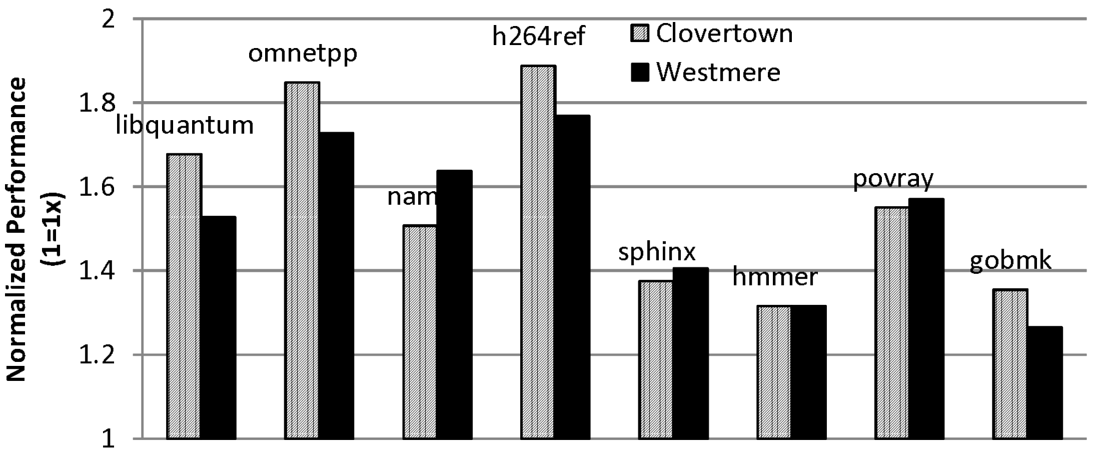


Figure 29: Speedup with cache balancing

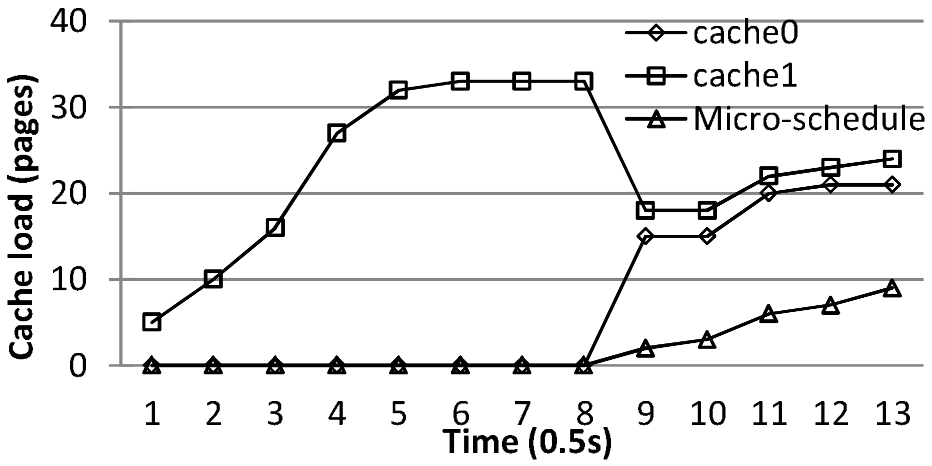


Figure 30: Cache-unification for two-region memlat

disturbance by the presence of other cache-intensive codes.

Conversely, performance can be improved for cache-intensive codes by giving them access to multiple caches, termed cache unification. The initial effects of cache unification on cache loads for the simple memlat micro-benchmark on machine Clovertown are shown in Figure 30.

The first half of the figure shows unbalanced cache loads, and the second half shows balanced loads plus micro-scheduling. Similar results are obtained on machine Westmere and for brevity, are not shown here. We evaluate the performance implications of such actions in more detail below.

2.5.3 SPEC Benchmarks

Figure 31 depicts measured results for experiments that assess the memory access pattern of the SPEC 2006 benchmark, which is obtained by collecting their access bit history over a test run (see Figure 26). From these runs, it is clear that libquantum and povray have simple access patterns, resulting in very stable regions, whereas sphinx and namd regions are more dynamic. This is verified by Figure 32, showing just a few regions in each cache for the first two, and a much larger number of regions for the latter two. Results obtained on machine Westmere are consistent, except that it tends to have bigger region sizes due to its higher performance (not reported for brevity).

Considering the access patterns depicted in Figure 31, these measurements show that the regioning methods correctly identify the memory region-based execution behavior of these codes, where e.g., libquantum and povray have few regions while namd and sphinx have many regions.

Another set of results in Figure 32 (the graphs on the right) de-pict the cache load (sum of region sizes for each cache) imposed by these codes. First, note that in the case of namd and sphinx, cache balancing succeeds in balancing both caches. This is in part because the number of regions for these codes is relatively large, which then permits the cache balancer to advantageously pack these regions into caches. In contrast, libquantum and povray show poor cache balancing, in part due to their small numbers of regions. Second, and as shown in Figure 33, successful cache balancing always improves performance, with an almost 15% gain for the Sphinx benchmark.

The outcome from these experiments is that regioning and cache balancing result in performance improvements even when the number of micro-scheduling actions is high. In fact and as shown in Figure 34, improved performance is seen even for very large numbers of micro-scheduling actions, e.g., the 10% improvement seen for namd on machine Clovertown is attained with up to 2000 micro-scheduling actions

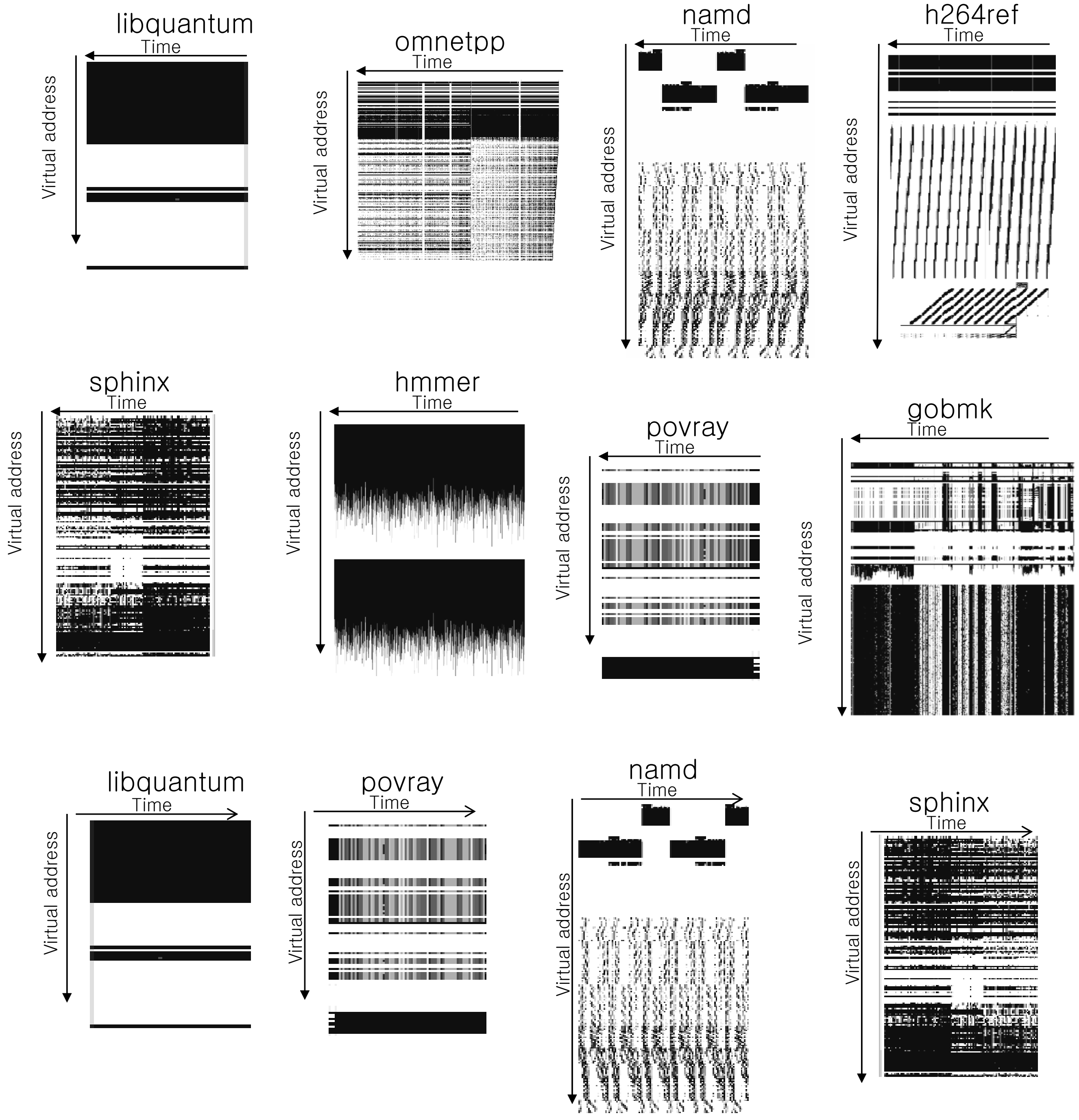
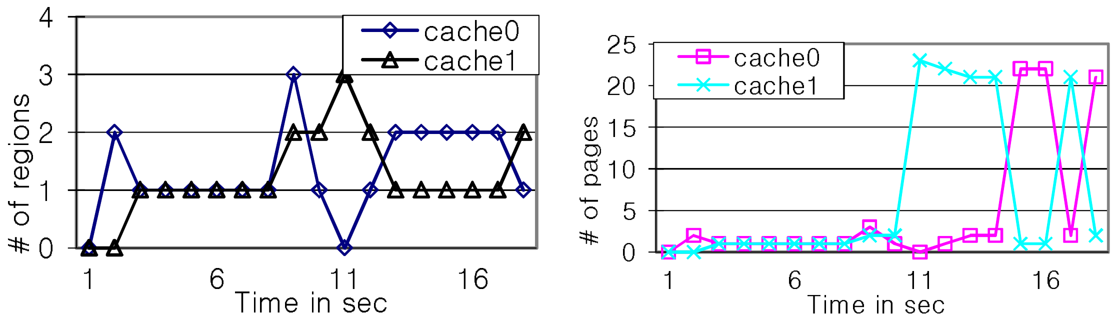
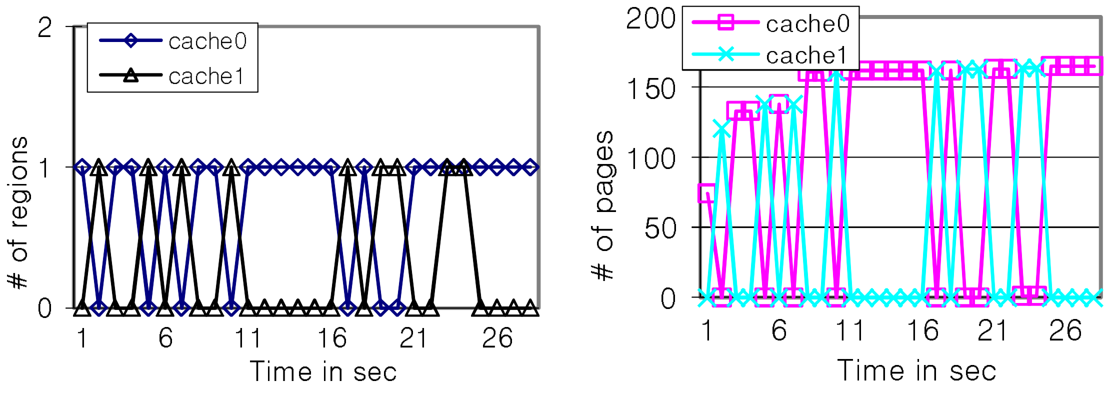


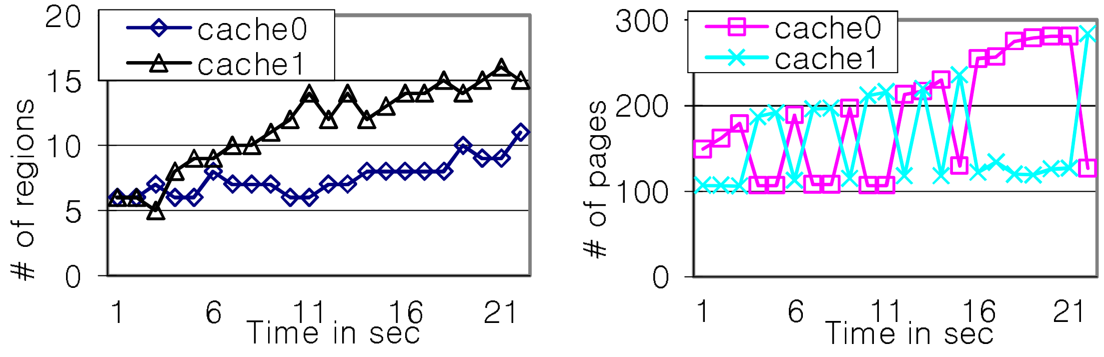
Figure 31: SPEC 2006 memory accesses



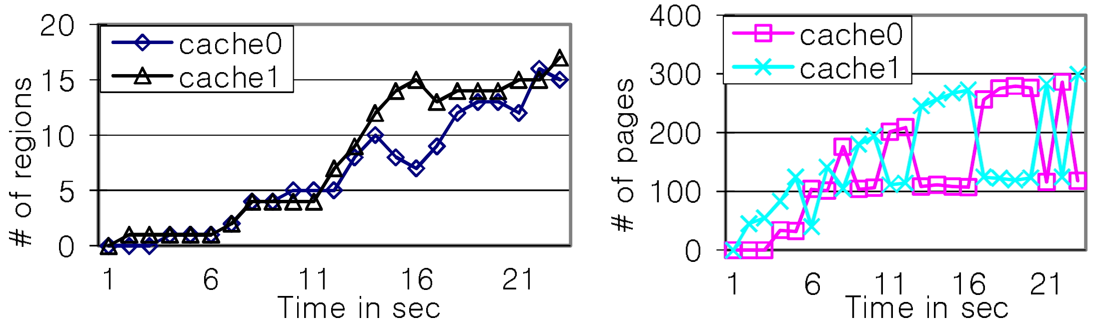
(a) libquantum



(b) povray

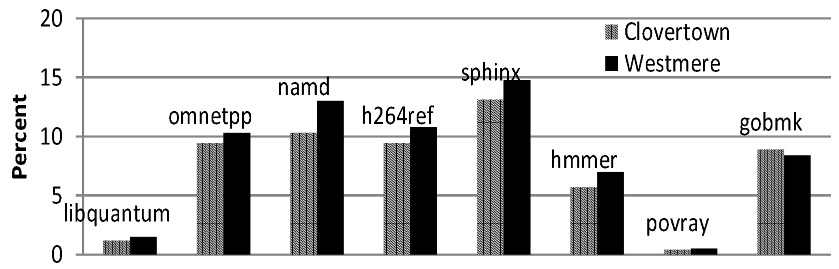


(c) namd

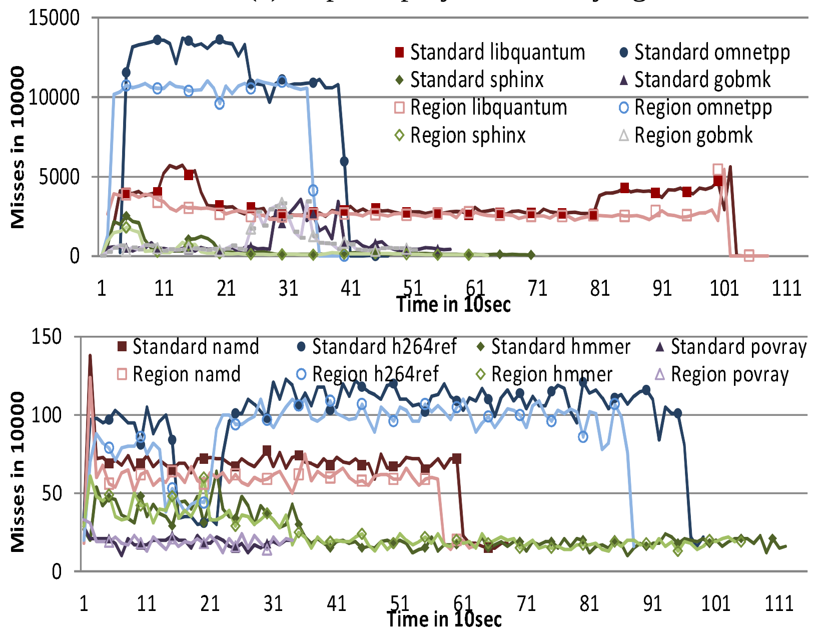


(d) sphinx

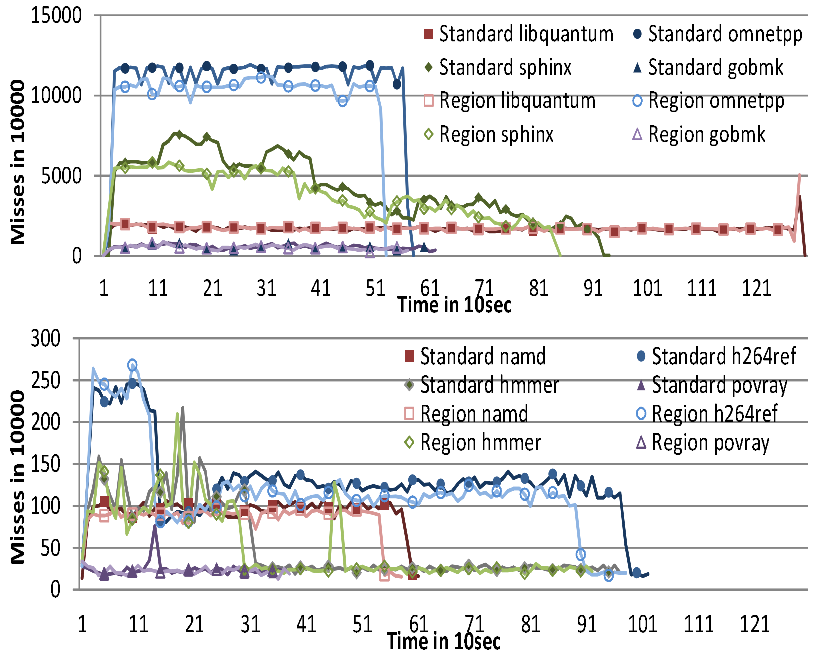
Figure 32: SPEC 2006 regioning. Regioning threshold = 100us on Clovertown



(a) Speedup by cache unifying



(b) Cache misses for Westmere



(c) Cache misses for Clovertown

Figure 33: SPEC benchmark cache unifying

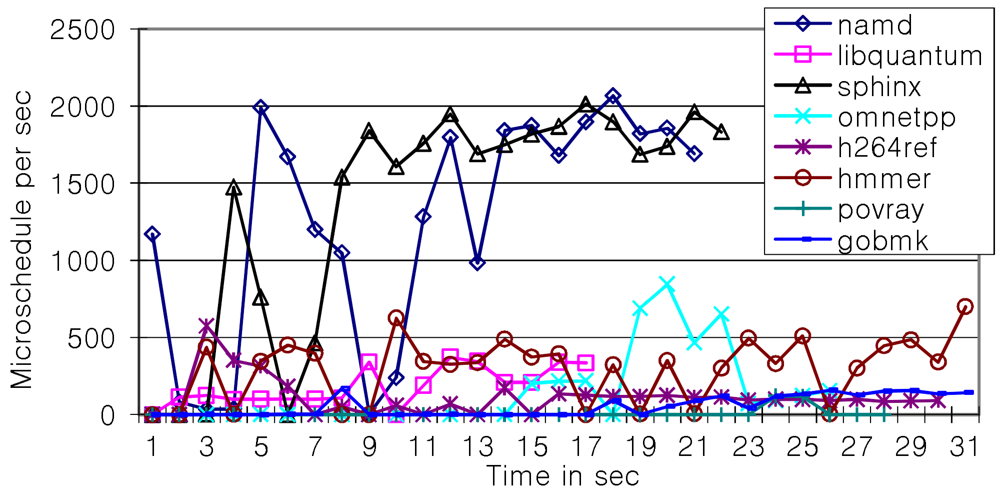


Figure 34: Micro-scheduling rates on Clovertown

per second! Further, Westmere shows better performance due to its cheaper micro-scheduling. The measured cache misses in Figure 33 demonstrate why this is the case. In many workloads, such as omnetpp, sphinx, h264ref, the measured cache miss rates are lower with region-based scheduling.

The fact that performance benefits are seen even with high micro-scheduling rates (up to 2000 context switches per second) is a key result of this research. This demonstrates that given the rela-tively low cost context switching on modern architectures, there is an almost overwhelming importance of caching to program per-formance. We view results like these as an important motivation for carrying out and continuing our research into memory- and cache-centric methods for processor scheduling.

We also note that these SPEC-based results are consistent with the memlat-based ones shown in Section 2.3, thereby demonstrat-ing that the potential behaviors we diagnosed with the memlat micro-benchmark are realistic in that they occur in actual codes. From the memlat-based diagnostic measurements, we also note that unduly high micro-scheduling rates can reduce or eliminate the potential performance gains derived from cache unification. This places constraints on the granularities of regions and region mappings that must be observed and taken into account by region

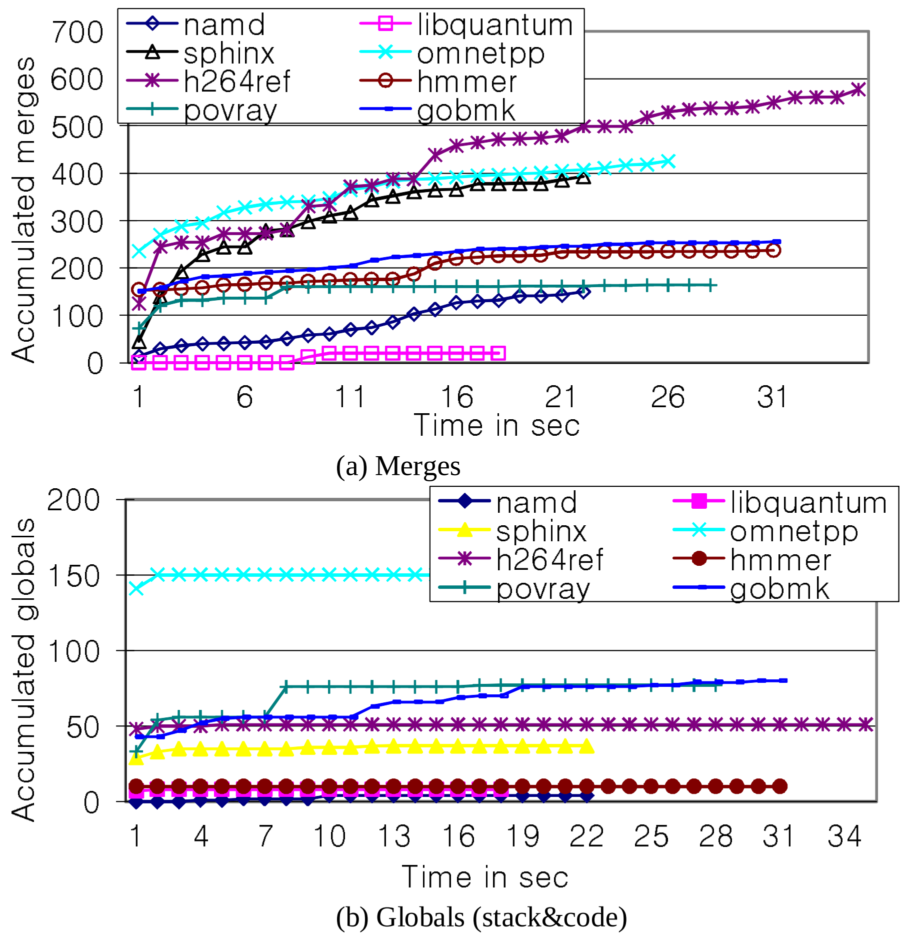


Figure 35: Regioning details (Clovertown)

scheduling.

Finally, the results in Figure 35 confirm that the bottom-up ap-proach to regioning used in our research is viable. First, since we start with many small regions, initially, there are many merge actions, but second, there is sufficient stability that the number of merges quickly subsides, along with the number of re-regioning actions, as evident from the number of global regions seen in these codes. We deduce that memory regions are sufficiently stable to warrant their runtime detection and use, even without special hardware support for doing so.

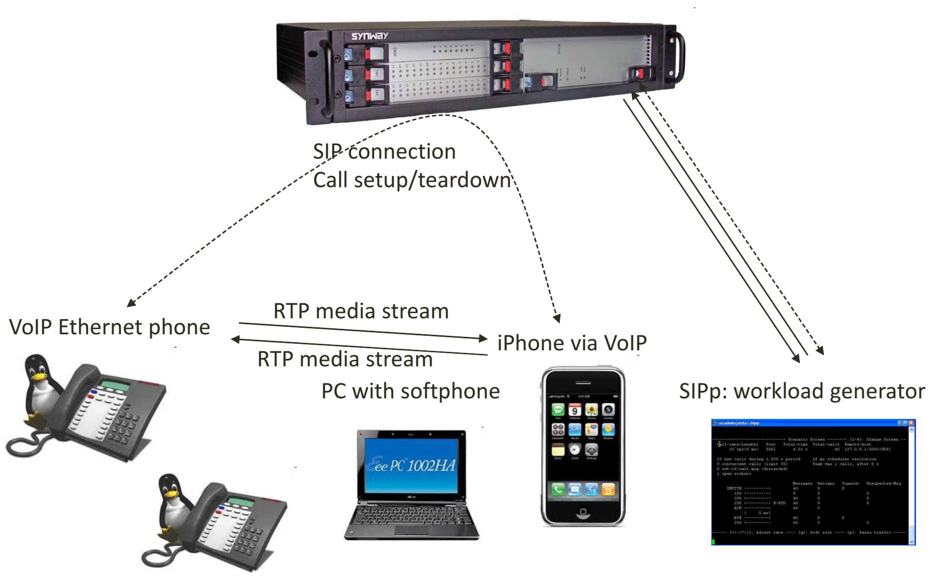
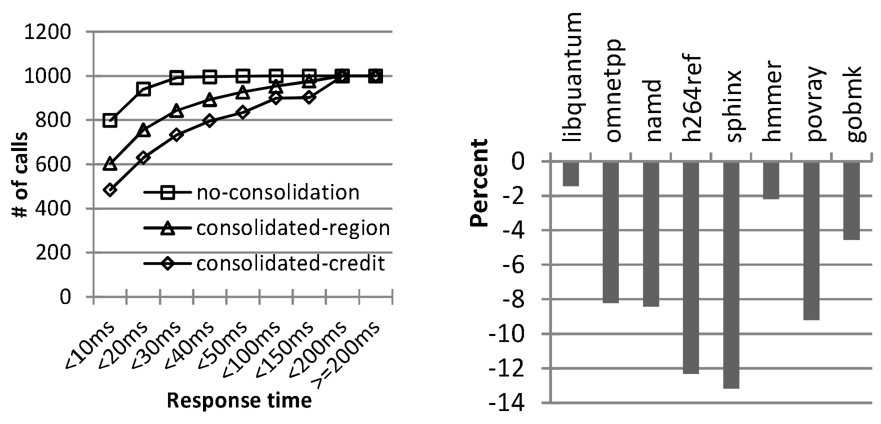


Figure 36: Asterisk server and clients

2.5.4 Media Benchmark

Region scheduling can help improve performance, as shown in Section 2.5.3, but it can also improve other important metrics, such as noise [26] for parallel codes or timing perturbation for real-time applications. We demonstrate the latter on machine Clovertown by measuring the response times seen for high performance IP telephony software, Asterisk [25].

Asterisk is a complete IP PBX, comprised of a voice commu-nication server featuring voice mail, conference calling, interactive voice response, etc. It supports VoIP protocols such as SIP, MGCP, and H.323. It acts as a signalling server (SIP server) and a media server. When it handles signalling, it deals with call set-up/teardown, etc. When it is used as a media server, it takes a voice stream, processes it (including transcoding, if needed), and then sends the stream to the recipient. Figure 36 shows the configuration for Asterisk. This real-time media server requires low latencies to process voice streams, and it requires that those latencies remain within certain upper bounds to protect voice quality.



(a) Respnse time (b) SPEC cost

Figure 37: Media server+SPEC consolidation on Clovertown

Using the SIPp traffic generator for the SIP protocol, we run experiments exercising the system at 10 calls per second with RTP traffic. Signalling is initiated from SIPp, and an 8 second pcap file (RTP stream of G.711 encoded) is sent to the media server after call establishment. The call hold time is 10 seconds. The parameters above imply that there are at most 200 RTP streams flowing into the media server at any point of time (100 streams from the caller and 100 streams from the callee).

2.5.5 Cache Sharing

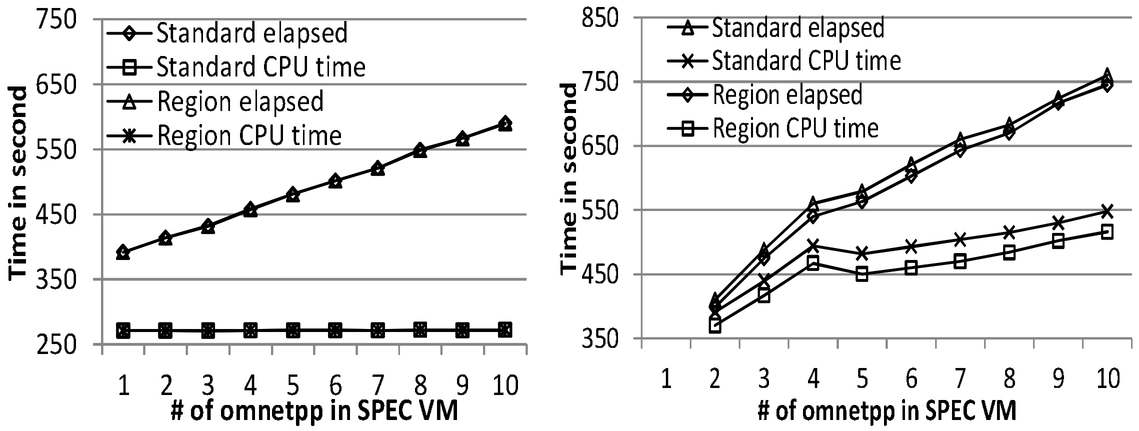
To demonstrate the utility of cache balancing, the VOIP experiment uses unfair policies that offer additional cache space to a preferred virtual machine. This is particularly important, of course, when there are multiple applications that share access to the platforms CPU and cache/memory resources.

Figure 37 shows the media servers improved response time on machine Clovertown when it is the only application running (denoted no consolidation), when no region scheduling is used (denoted consolidated-credit), or when region scheduling is employed and the media server is the preferred VM (denoted consolidated-region). Results show that the servers response times are much more consistent for the region-based vs. credit-based scheduling approaches, and both are worse, of course, than the non-shared scenario. The figure shows the cumulative number of calls with various durations observed during the runs, with a flat line being best. We also note that the overall average response time is 7.765ms for no-consolidation, 19.415ms for consolidated-region, and 32.535ms for consolidated-credit, re-spectively. The almost 40% improvement in the average response time seen for the server comes at a moderate cost for the other applications running on the platform, with an up to 15% detriment observed for the sphinx code. Degradation occurs because the cache balancer is instructed to provide additional cache to the media server, which is known to be cache-sensitive.

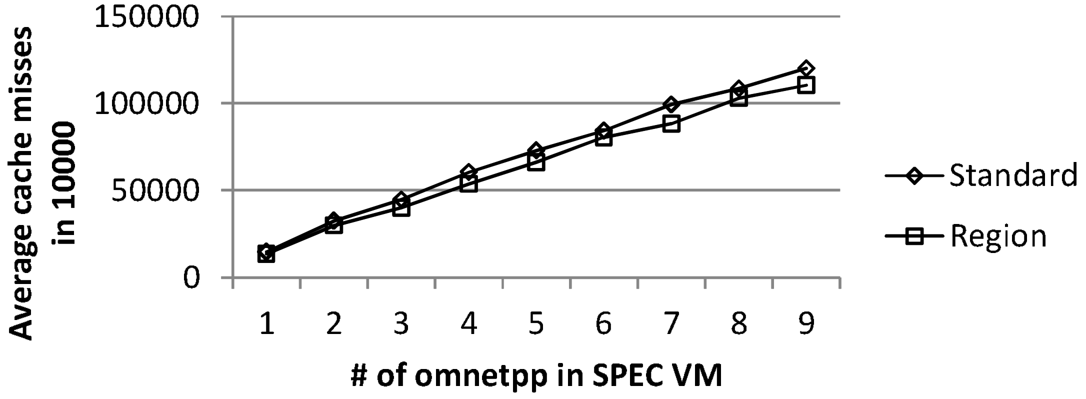
2.5.6 Reducing Noise for Parallel Codes

We next explore the use of region scheduling to protect parallel codes (e.g., OpenMP codes) running on a shared platform from each other and/or from the effects other codes on the same platform may impose on them. This is particularly important as we move toward many-core systems where the platforms on which parallel simulation computations take place will be shared with other applications (as in consolidated and cloud computing systems) or will be shared with additional codes that analyze simulation output data as it is being produced [35]. One could, of course, strictly partition nodes and their caches, but an approach like region scheduling that explicitly understands memory usage and can better isolate codes from each other, as evident from the results shown in the previous section, should result in improved levels of node utilization and permit richer and more finer-grain ways of using and sharing the node resources of future machines.

We use two virtual machines one virtual machine running a ray tracing parallel workload (PAR VM) and other a SPEC om-netpp workload (SPEC VM). Figure 38(a) shows the measured (worst) elapsed time and (average) CPU time for each virtual machine. In case of raytrace, elapsed time increases as more workload is added to



(a) Performance for two virtual machines (Westmere)



(b) Cache misses (Westmere)

Figure 38: Parallel code VM consolidated with SPEC VM

the SPEC VM. However, its CPU time stays constant. This is because this parallel code does not reuse its data, so is not impacted by cache contention. Meanwhile, the omnetpps performance suffers from sharing cache with PAR VM. Region scheduling improves the SPEC VMs performance by mostly isolating PAR VM onto one cache while SPEC VM runs on the other cache. Figure 38(b) shows their average cache misses. In general, with region scheduling, we observe 8.39% less cache misses in Figure 38(b).

2.6 Related Work

The importance of efficient cache usage is well known [12, 13, 14, 15]. There are hardware approaches to cache partition (e.g., based on utility metrics [23] or spill/receive [24]) and software-based methods. The ones proposed in [16, 17] are similar to those used in our work, but these simulation-based results are focused on NUCA caches [12], whereas we contribute a general framework for and implementation of policies for cache management for realistic multi-cache, multi-core platforms.

Other cache-aware schedulers [5, 8] use thread migration and matching, and [6, 7] use page coloring or guided page allocation to partition shared caches, whereas we use page-table-based page-level affinity and microscheduling. Since our methods are imple-mented at hypervisor level, they can be used without modifying operating systems. Further, we go beyond earlier results to deal with multiple rather than the single caches addressed in prior work, and for such multiple caches, we go beyond cache partitioning to also support cache unification. Finally, we can estimate cache loads, since we track region working sets [10, 11].

Recent work at MIT has commonalities with our work, using a synthetic directory workload [31] as a demonstration. In that research, ideas similar to ours [32, 33, 34] are implemented in hardware, using instruction-level execution migration. We differ in that we extend the idea to cover all of a systems memory, and we do so in the VMM in order to make the functionality accessible to arbitrary unmodified applications and systems, without requiring any hardware changes.

Affinity scheduling [9] constitutes early work motivating the importance of caching for high performance codes. Our work can be thought as a next step in such work page-level affinity. We have identified and demonstrated how this page-level affinity could be used.

The term region scheduling also appears in [36], but that work has nothing in common with what is presented in this chapter.

2.7 Conclusions And Future Work

This chapter introduces a memory-centric approach to managing the resources of multi-core platforms, motivated by the ever-increasing importance of memory and cache resources (and their efficient use) in multi-core architectures. Indeed, we show results where improved performance is gained due to superior cache usage even at the cost of relatively high rates of context switching (e.g., up to 2000 micro-scheduling actions per second). Intuitively, this is because it is preferable to move the computational entity to where its memory is vs. moving the memory (i.e., cache lines) to where the entity currently runs.

To realize cache-centric scheduling, we introduce the novel notion of memory regions and then develop system-level support for dynamically determining these regions for mapping them to caches so as to optimize program performance. The resulting region framework is realized as a software layer in the Xen hypervisor, and beyond determining and mapping memory regions to caches, its additional task is to ensure that the entities touching memory regions are run so that region-to-cache mappings are preserved, i.e., a process is run only on a core associated with the cache in which its region is currently mapped.

Region scheduling could benefit from additional hardware support. For instance, execution migration by hardware [32, 33, 34] could reduce micro-scheduling overheads to only 100 cycles [34], thereby further broadening the usefulness of region-scheduling to applications. Further, there may be hardware-level opportunities to exploit the information about the behavior of an address space in terms of its region accesses and working set size. A particular opportunity is to use such inputs to affect the cache eviction policy used by hardware, i.e., to select victims for eviction. For example, once a region is unmapped from a cache, existing cache lines from the regions are ideal victims because they will not be accessed through that cache until opening region.

Other options include (i) not to evict cache lines from shared pages, such as those containing library codes, or (ii) to exploit the memory reference patterns detected by region scheduling to choose as victims one-time data, e.g., one-time data such as the inputs or outputs produced by codes. Finally, (iii) rather than pursuing hardware methods for cache partitioning, one could exploit region scheduling for soft cache partitioning. Such partitioning can be used to give unfair advantages to certain codes, or even to dynamically resize codes cache sizes to adjust them to their current working set sizes. On asymmetric multi-core architectures, this would make it possible, for instance, to isolate small workloads onto a smaller cache while giving most other cache capacity larger workloads.

This chapter clearly demonstrates the promise of memory-centric scheduling, but there are several limitations in the current region framework: (i) to understand parallelism in multithreaded applications remains future work; (ii) if no region is detected, there are costs but not benefits this should be addressed; (iii) the OS kernels in guest operating systems remain undifferentiated global regions this is the subject of the next chapter.

CHAPTER III

DISAGGREGATED OS SERVICES

3.1 Introduction

The previous chapter introduces the region scheduler and the bottom-up detection of regions, the latter monitoring the guest VM's memory access patterns. The methods described are applied to user space, not kernel space, because their effectiveness derives from monitoring the behavior of applications to determine appropriate page to region mappings. Because the OS kernel space is shared by all processes running in the virtual machine, the page access behavior seen in the OS kernel is an aggregate behavior of all currently running applications. It is unclear to what extent such an aggregate can lead to useful region creation. Further, since all applications – wherever they run – share access to the single kernel space, it is also unclear how detected kernel space regions should be mapped to caches, along with the regions of the applications making the kernel calls that in turn, give rise to kernel-level regions. Stated intuitively, placing a kernel region onto a certain cache will cause a microscheduling action for any application's kernel call occurring on some other cache; with frequent kernel calls, this will give rise to an inordinately high frequency of microscheduling actions. Finally, for kernels with their complex interleaved internal code paths and compactly laid out data structures, the automated regioning approach is limited and inhibited by its relatively large 4KB page granularity. Specifically, while regions are composed of 4K page sized entities, kernel data structures may not fit these 4K sizes and/or may be co-located in the same 4K pages[129]. Needed is an additional concept to assist us in kernel-level regioning.

In this chapter, we explore how the memory region approach can be applied to the OS kernel and investigate its implications. To do so, several challenges have to be overcome. First, the region scheduler for application-level data is based on page-to-cache mappings, with 4KB pages. This means that any objects or cache lines in the 4KB page are treated as related objects. This false sharing phenomenon is well-understood, and solving it would require (i) regioning at the cache-line level, which is not feasible with current hardware or (ii) the redesign of kernel data structures, which is not desirable due to the resulting lack of generality of the approach and the ensuing reduced maintainability of thus redesigned OS kernels. To address these issues, we (i) offer new functionality – object to region mapping – and (ii) apply this functionality only to select kernel data structures and services, thereby demonstrating the utility of regions and region scheduling for kernel level services, but without the automated regioning used for application data.

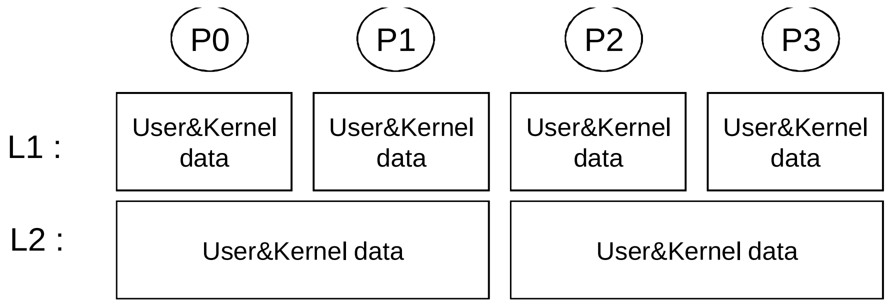
3.1.1 Kernel Disaggregation

Region scheduling has the interesting and important property that by placing certain kernel data structures onto certain caches, the execution of kernel services touching those structures is determined to run on the cores associated with those caches. As shown with the layout of user and kernel data depicted in Figure 39, kernel-level region scheduling, therefore, can create the notion of user vs. kernel cores. In Figure 39(b), P0 and P1 are user cores, while P2 and P3 are kernel cores. This effectively specializes the cores to run certain software functions, regardless of the underlying hardware, e.g., whether it is a homogeneous or a heterogeneous ISA architecture. The outcome is what has been termed 'kernel disaggregation' and in past research, such functional disaggregation has required OS kernel redesign and/or new kernel mechanisms assisting disaggregated kernel operation. Region scheduling can be used to disaggregate kernel functionality without any changes to kernel code, as demonstrated in this chapter.

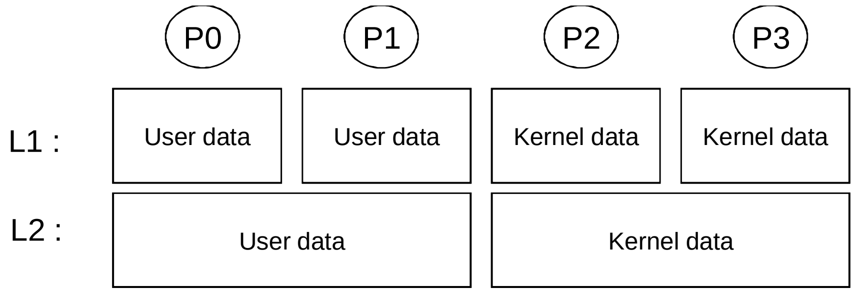
Kernel disaggregation refers to the separation of specific kernel services or functions in order to direct their placement to or confine them to certain machine resources [132]. The approach has been shown useful for reducing OS ‘noise’ in high performance systems[130], and it is the defacto standard for hardware platforms on which certain OS services are statically mapped to run on specific hardware components. An example of the latter is the mapping of communication functions to smart network interface cards (NICs), sometimes also referred to as ’communication offloading’ [131].

We offer the following general motivation for the utility of kernel disaggregation. Consider modern cloud infrastructures [113] in which many VMs share underlying physical machine resources. Current sharing methods implemented in say, VMware vSphere [111] or Citrix XenServer [112], have explicit mechanisms to fairly share CPUs, physical memory, [109, 110]. They do not, however, consider fairness in how CPU caches are shared, despite the known importance of CPU caches to application performance. The reason, of course, is that CPU caces remain invisible to system software. Using regions and region scheduling, we can make CPU caches into a first class schedulable resource considered in virtualized systems. Kernel disaggregation, then, is a useful and interesting side effect of kernel-level regioning.

As stated above, however, kernel-level regioning is challenging and difficult to automate. The approach taken in our work is to forego automated regioning for kernel-level data and instead, rely on kernel developers to identify and mark – not redesign or rewrite – kernel data structures suitable for regioning. The outcome is an approach in which developers identify object to region mappings, which are then used by the region scheduler to appropriately map regions to caches. Stated with an example, the basic idea of region scheduling method is to place working sets onto each cache and move computations to those cores. Figure 39(b) shows how memory blocks are placed into caches using this scheme. With current CPU-centric



(a) Traditional CPU-centric scheduler



(b) Memory-centric scheduler

Figure 39: User/Kernel cores and caches.

scheduling methods, i.e., without kernel level region scheduling, the caching hardware mechanism blindly places data from both user and kernel space onto the same caches, as shown in Figure 39(a). In addition, the load balancer will migrate tasks to other caches, creating duplicate cache lines. With the memory-centric scheduling approach realized with region scheduling, data is simply mapped onto each cache first, and then the tasks are scheduled onto appropriate PCPUs. This results in reduced cache line duplication, and can be used to confine tasks to certain caches.

To explain kernel-level region scheduling, we next briefly review the concept as realized for application-level data, followed by a description of how it can be applied to kernel-level data.

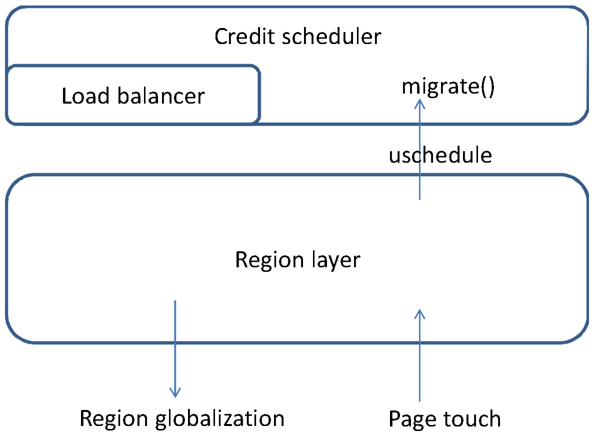


Figure 40: Region layer and CPU scheduler

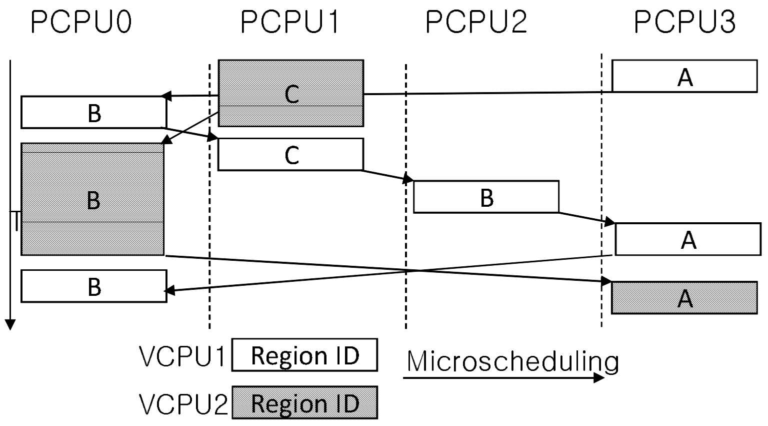


Figure 41: Region scheduling

3.2 Memory-centric Scheduling

3.2.1 Region scheduler

As explained earlier, region scheduling is implemented in Xen, which has its own VCPU scheduler, the credit scheduler. The region scheduler (or the region framework), then, is implemented as a layer below the credit scheduler. It is responsible for region-to-cache mapping, meaning that is is able to map a region – a set of physical pages – onto some target cache.

Figure 41 illustrates region scheduling for two VCPUs. PCPU0 and 2 share the first LLC (last level cache) which holds the memory block B, while PCPU1, 3 share the second LLC which holds the memory blocks A and C. VCPUs are scheduled based on which memory blocks they wish to access. So, for example, a VCPU would be scheduled onto PCPU0 or 2 if it accesses B. It would be scheduled onto PCPU1 or 3 if it accesses A or C.

3.2.2 Regioning

Region scheduling is governed by regioning page tables, one per process. In that table, every entry is closed by default, meaning that an access to that entry generates a page fault visible to the hypervisor. The regioning page table is the basis on which regions are formed for each process. For a process to enter the regioning phase, we simply perform a switch to its regioning page table, which then causes the subsequent sets of page faults that permit us to assess its page access behavior. The outcome is low overhead for each process' entry into a regioning phase. For multithreaded applications, one could implement per-thread regioning page tables, with consequently increased complexities and space overhead. The current implementation uses a single page table shared by all threads for the given process. Each thread enters the regioning phase at different times, avoiding overlap. This means that regioning can be performed only one thread at a time, thus extending its duration. This choice is suitable for longer-running, somewhat stable applications or application cohorts, with an alternative solution required for highly dynamic, time-varying codes. We note that there are additional challenges for regioning, including how to deal with other dynamic behaviors, such as those caused by interrupts, changes to address spaces, etc.

3.3 Guest-defined Region

The region scheduler defines the region-to-cache mapping, with regions detected by the regioning algorithm. To deal with the fact that particularly for kernel-level data, a single region may contain un-related objects – due to the 4KB page sizes in our system

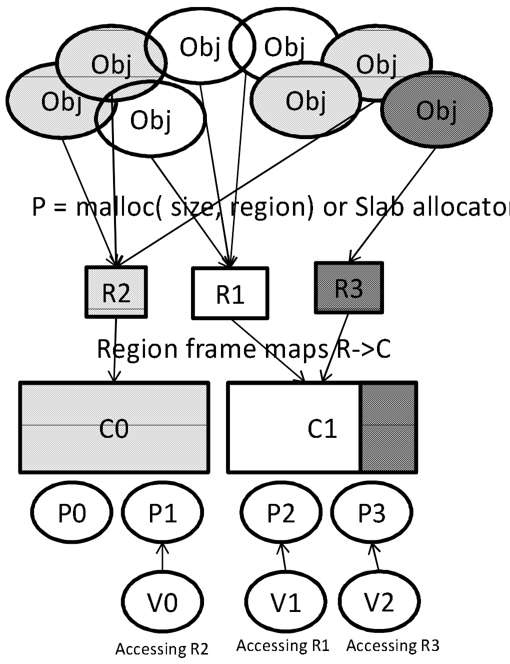


Figure 42: Guest-defined regions

– we introduce methods that permit a guest to identify regions, relying on specifications stated by system developers. This method is particularly useful if regions are carefully placed to avoid false sharing or if there is compiler support with which data structure mappings to pages can be controlled. For some managed languages, their runtimes can handle such tasks automatically, without the programmer-based annotations used in our work [126]. Additional useful support for controlling object to region mappings might be provided by garbage collectors or heap managers, which routinely free or move objects from one place to another.

In operating systems, one entity controlling object to region mappings is the slab allocator [127], its task being to cache frequently used objects. This naturally creates many regions of related objects, making it a good candidate for a developer-defined, per guest, kernel-level region. For example, the inode cache is a key data structure of the file system, and the slab allocator manages these inode caches.

In the remainder of this chapter, the region framework is extended to embrace the

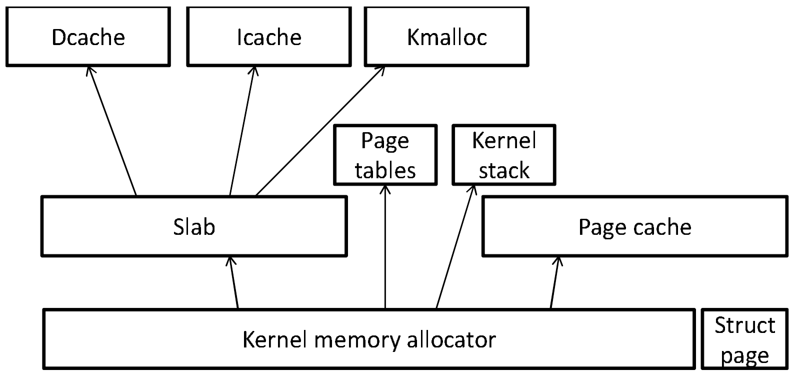


Figure 43: Linux memory and slab allocator

object-to-region mappings needed for creating regions for select kernel data structures, thereby indirectly configuring the kernel functions using those structures to run on designated processor cores. Such OS service disaggregation is demonstrated by applying the guest-defined regions to select Linux kernel data structures and functionality.

A guest-defined region is a region in which a guest specifies which pages belong to what region. The concept applies to both user-level and kernel-level data, but in this chapter, it is applied to kernel data only. In either case, the guest maps objects to regions by annotation. For example, the malloc() call can be extended to have an additional parameter that specifies a region. Then, the region framework maps the regions onto caches. Or a kernel developer can select kernel data structures, annotating the kernel functions using them and/or interposing the calls made to these structures with annotated calls. In any case, note that region-to-cache mapping is still done by the region scheduling framework, which may globalize and localize certain regions, if necessary. Each VCPU will then be scheduled to the correct PCPUs. For example, in Figure 42, if the R2-to-C0 mapping is given, then V0 runs only on P0 or P1 to access R2's objects. Similarly, V1 and V2 run on P2, P3 to access R1 and R3, respectively.

The guest-defined region API is implemented with the hypercalls in Table 3. First,

```
int add_guest_region(void); // returns Rid
void del_guest_region(int rid);
void add_page(mfn, order, rid);
void del_page(mfn, order, rid);
```

Table 3: APIs for guest-defined region

the two functions add_guest_region() and del_guest_region() create or destroy regions. Each returns or takes the region ID (rid) as region identification. The other two functions, add_page() and del_page(), add or delete physically contiguous pages to the given region.

The creation of guest-defined regions may be viewed as manual regioning. It is efficiently done at allocation time. Once specified by the guest, the hypervisor maps/manages the placement of regions to caches. Creating and using guest-defined region is simple and beneficial:

- The annotation effort is minimal. For instance, we added only a few (less than 10) lines to linux/mm/slab.c using these hypercalls, with negligible runtime overheads incurred from the use of these calls.

- Among many object caches, we chose the key data structures ext3_inode_cache, ext3_xattr for the file system subsystem and skbuff_head_cache, skbuff_fclone_cache for the networking subsystem, thus making it easy to disaggregate these two important sets of functionalities to certain CPU cores.

3.4 Load Balancer Conflict

While traditional CPU-centric schedulers focus on CPU utilization, the region scheduler treats the memory blocks accessed by the current thread (or VCPU) as first class schedulable entities. When the thread accesses remote memory blocks, it is microscheduled to the processors associated with the cache where the target memory blocks reside.

A potential interesting issue with region scheduling is microscheduling leading to short time slice lengths. CPU schedulers deal with time slice lengths as follows. Consider a web server generating dynamic pages. For such a VM, the credit scheduler increases time slice lengths as more CPU is consumed to generate dynamic pages. With cpu-boundness=100, over 30% of the CPU time is consumed with time slices of 30ms, which indicates the VCPU is preempted only by the credit scheduler. (The credit scheduler uses a max. 30ms time slice by default). Further, we observe most other short time slices to be consumed by network processing activities. This is natural because of the web server's workload need for more CPU time. With region scheduling, however, VCPUs' time slice lengths do not increase, because of the frequent microscheduling actions taken for them. In other words, with region scheduling, a single VCPU does not accrue sufficient CPU cycles on any one core to prompt the CPU scheduler to increase its time slice length. Instead, with the VCPU running across multiple cores in a distributed manner, inaccurate accounting leads to inaccurate inputs to the traditional credit-based scheduler. We respond to this problem by modifying the existing credit scheduler to better understand a VCPU's time consumption when microscheduled. Specifically, time accounting is refined to use nanosecond time resolution based on processor cycles rather than using traditional tick-based time (Xen adjusts this value to cope with frequency drift). The result is finer-grained time accounting and more accurate time measurement for VCPUs.

Another issue with region scheduling is its potential conflict with the traditional load balancer on CMP/SMP platforms. The traditional load balancer tries to reduce CPU idleness across the platform. However, in the memory-centric approach, sometimes, idle CPUs are natural. Therefore, the load balancer that prevents CPUs from being idle by migrating VCPUs may conflict with the region layer that may keep some CPUs idle.

A load balancer blindly moving VCPUs may cause 'spurious' microscheduling.

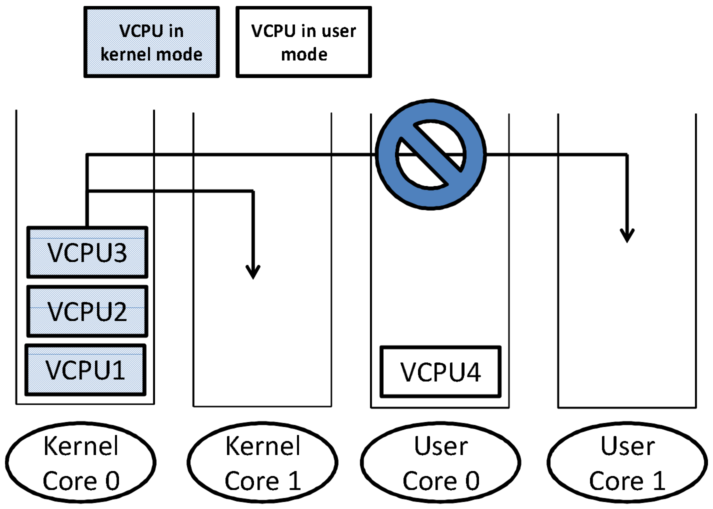


Figure 44: Load balancer conflict

For example, in Figure 44, VCPU1, 2, 3 can move to kernel core 1, but not to user core 1. If it moves to the user core, then it will immediately invoke microscheduling back to the kernel cores. This adds unnecessarily two microscheduling and such spurious microscheduling may impact performance significantly when it occurs frequently. Even when VCPU3 moves to kernel core 1, user core 1 is left idle and this cannot be resolved until some VCPU switches back to user mode. So, in this model, when we have unbalanced workloads in each mode, it is natural to have idle cores and therefore, lower CPU utilization.

One easy approach to prevent the spurious microschdulings is to disable load balancing across cache boundaries. Although it still allows load balancing within cache-sharing cores, it easily result in lower CPU utilization. However, this approach may be okay with ample CPU resources where there is almost no needs for load balancing.

Rather than just disabling load balancing, our approach is to modify the load balancer so that it carefully chooses VCPU candidates for migration. The basic idea is to mark the VCPU that is just microscheduled so that the load balancer does not pick that VCPU unnecessarily. Figure 45 depicts this scenario. Assume that the VCPU3

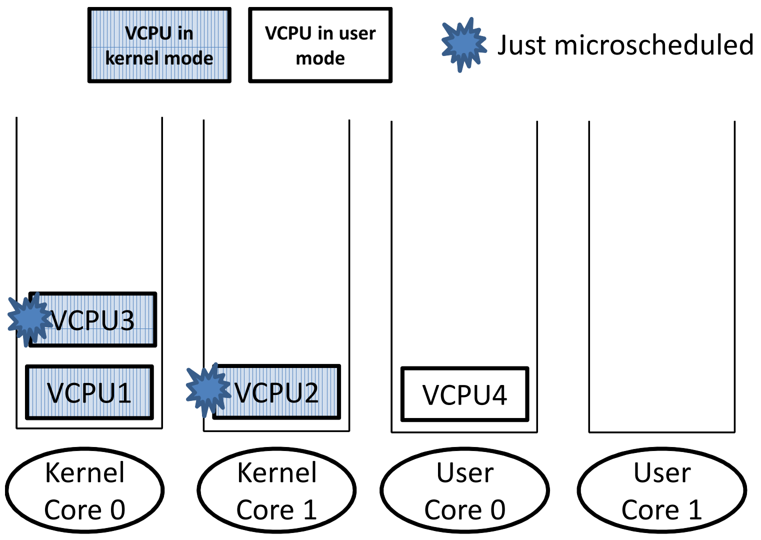


Figure 45: Load balancer conflict

caused a page touch and just got microscheduled to Kernel Core 0. At this point the VCPU3 is about to access its kernel data structure and it does not make sense to move it back to the user cores since it would again cause another microscheduling back to kernel cores. Note that the VCPU3 is just got moved to Kernel Core 0 so it never had a chance to resolve its accesses to kernel data structures. So, we can mark VCPU3 as an "just microscheduled" then the load balancer can skip it. Once the scheduler schedules VCPU3 on Kernel Core 0 (after VCPU1 and VCPU2), the mark is removed because it is likely that VCPU3 will access to the wanted data. Once a time slice is given to VCPU3, it would be returned to the run queue with the mark removed. This guarantees that the microscheduled VCPU runs at least one time on the target core. At this point, it is one of the candidates for load balancing. i.e., The mark is removed when the VCPU is once scheduled on the core it is microscheduled to, and the load balancer skips VCPUs with the "just microscheduled" mark. Similarly, we can see that VCPU2 in Figure 45 is also just microscheduled because it is marked so, and we know that moving VCPU2 would be also unnecessary spurious microscheduling. Meanwhile, VCPU1 does not have the mark and it indicates that it had at least one time slice run on this core, which makes it a good candidate for load balancing.

Thus, VCPU3 and VCPU2 must have been running on user cores until they just get microscheduled to the kernel cores. Hence, the load balacner does not choose any 'marked' VCPU.

This removes most of spurious microscheduling successfully. However, sometimes some spurious microschedulings are still observed when a VCPU does not consume its time slice enough long and just yield its CPU core quickly. i.e., it sleeps frequently and loses its mark quickly, which eliminates the hints to the load balancer.

So, finally we do the mark removal only after some threshold CPU time. Giving 10ms of CPU running time before the removal of mark effectively reduced spurious microscheduling to a negligible level. This is because the VCPUs being microscheduled rarely spend that much time (10ms) on the cache-sharing cores. However, note that we use intact load balancing algorithm to the cache-sharing cores. Since they share the cache, they can freely steal work among themselves.

One noteworthy point is the relationship between VM consolidation and the load balancer conflict. While traditional CPU-centric schedulers are not impacted significantly with the number of VCPUs, the memory-centric scheduler may be affected substantially. For example, we would not see any lower CPU utilization if only 2 VCPUs are given to the virtual machine in Figure 44 and 45. The microscheduled VCPUs would always find some idle core in the target cache since the last level cache has 2 cores each. So it would not have to be queued into the run queue. In fact, this is the case for many cloud platforms. In cloud environments, the underlying physical machines have lots of CPU cores while the virtual machines are various mix of small to medium number of VCPUs. In such environments, it is less likely that the load balancer conflicts with the memory-centric scheduler because of ample CPU core resources.

Meanwhile, higher degree of VM consolidation also reduces the conflicts that may cause the lower CPU utilization because there are more chances to fill idle cores when we have more VCPUs in the system. On cloud environments, often over-subscribed systems have a high degree of VM consolidation and the memory-centric scheduler would easily find VCPUs to schedule, which prevents idle cores.

3.5 Micro-Benchmarks

This section demonstrates kernel disaggregation via the region scheduler using a simple web server example as an microbenchmakr. This would help us explain the approach and its implications. For web server evaluation, we use the popular web server, apache2, running on CentOS and Xen. Requests are generated at the client side concurrently by the apache benchmarking tool, ab. The concurrency level of 40 is given, and 40000 requests are processed in total. Web-static represents static files of sizes 10KB, 100KB, and 1000KB. However, the modern web servers often serve dynamic pages using dynamic languages such as PHP. Therefore, such dynamic page generation is emulated. Web-php represents such dynamically rendered pages and emulates CPU load using cpu-boundness parameter that is the number of pages to generate. It is basically the number of loop iteration, so 100 is ten times more computation than 10. We use cpu-boundness value 1, 10, 50, 100 to represent least, little, medium, intensive cpu-load for Web-php.

The machine has four cores, with two declared as kernel cores (kernel0, 1) and the rest (user 0, 1) as user cores. Kernel cores share one last level cache and user cores share a different last level cache. Therefore, the number of VCPUs is 4 by default. In addition, we will also take a look at the case of 2 VCPUs. The Figure 46 and 47 is Web-static, which serves static files and the Figure 48 and 49 is Web-php, which dynamically generates web pages using php interpreter. Each graph also has 2VCPU case for comparison.

The Figure 46 and 48 shows CPU utilization of each physical CPU cores in percentage. X axis is the name of cores where kernel0, kernel1 are kernel cores and user0, user1 are user cores. Y axis is the CPU utilization percentage and the box shows 25th percentile value and 75th percentile value, with the median value in it. The box has two lines at its head and bottom, showing the maximum and minimum values.

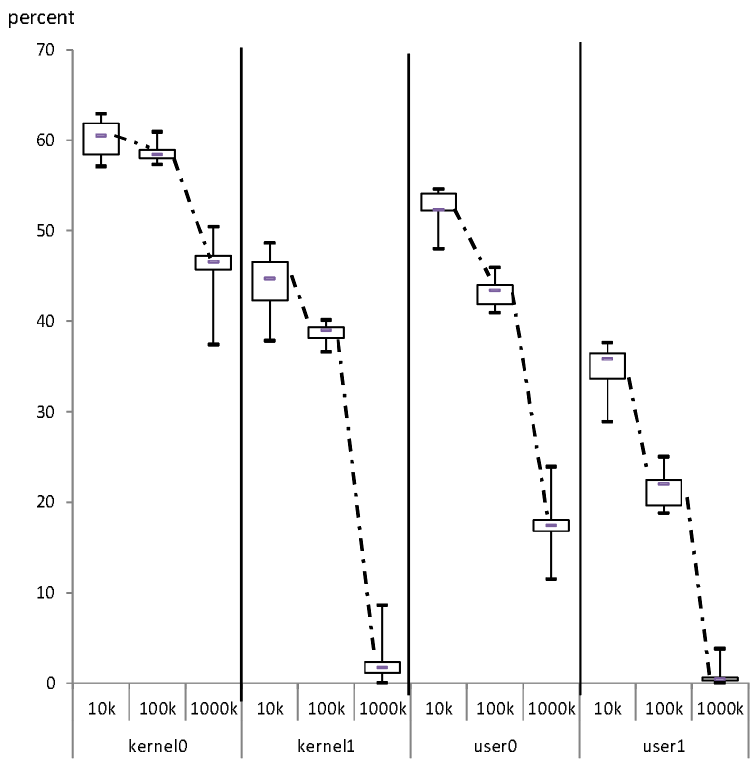
The Figure 47 and 49 is similar to CPU utilization graphs, but they show three statistics for the corresponding run, system call rate (syscall), microschduling rate (usched), and kernel-user mode switch (switch).

3.5.1 Kernel cores

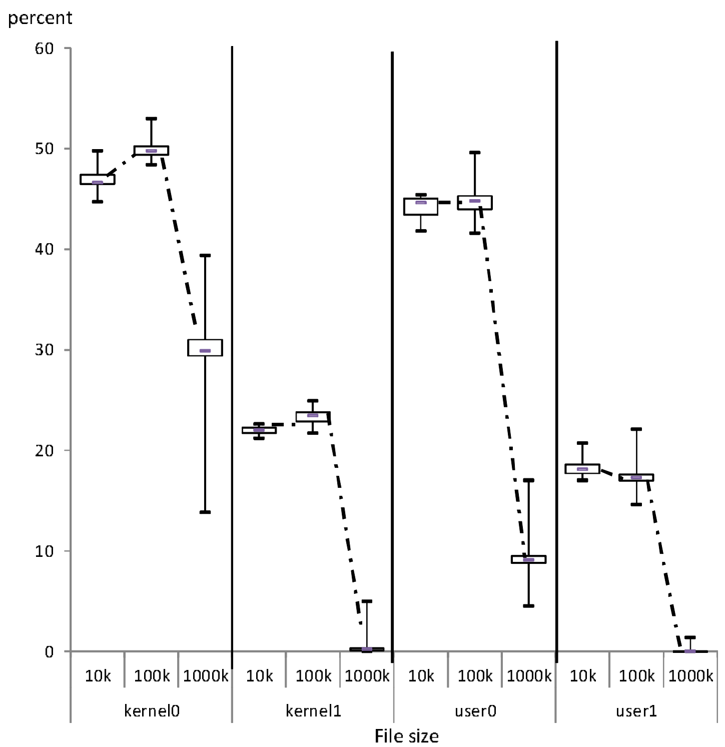
In Figure 46 CPU utilization graph for Web-static, we can see that the overall CPU utilization drops as the file size increases, because the network interface card handles most of the work. Also, if we compare the kernel space to user space, the kernel space activity is a bit more than the user space activity. (The total kernel space activity would be sum of kernel0 and kernel1 and similarly the total user space activity would be sum of user0 and user1.) At file size = 10k, the kernel-user ratio is roughly around 6:4. However, as the file size increases, user activity reduces sharply. At file size = 1000k, the kernel core is steadily utilized (see kernel0), while the user core's utilizations drops sharply. This is because the portion of network processing increases as the file size increases. We can see a similar trend with 2 VPCUs.

Figure 47 shows other statistics, the number of system calls, microscheduling, and kernel-user mode switches per second. They all drop as the file size increases, spending more time on network processing. The microscheduling rate is twice the system call rate as microscheduling happens at each system call entry and exit. "switch" shows the number of kernel-user mode switches per second, and this includes other kernel activities such as interrupt handling in addition to system calls.

Figure 48 CPU utilization graph clearly shows big portion of user space activity, because they are Web-php dynamic page generation. As more processing is done in user space to render the requested page, user cores get more and more active

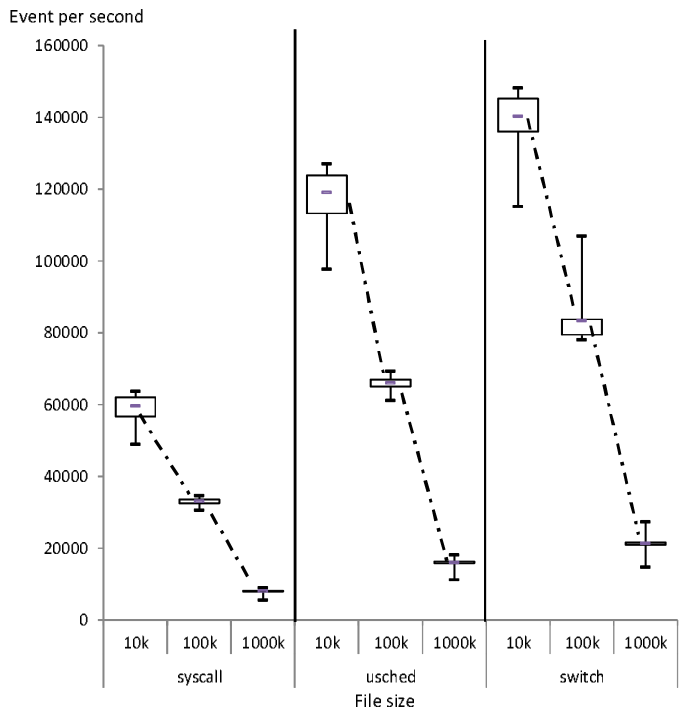


(a) 4vcpus

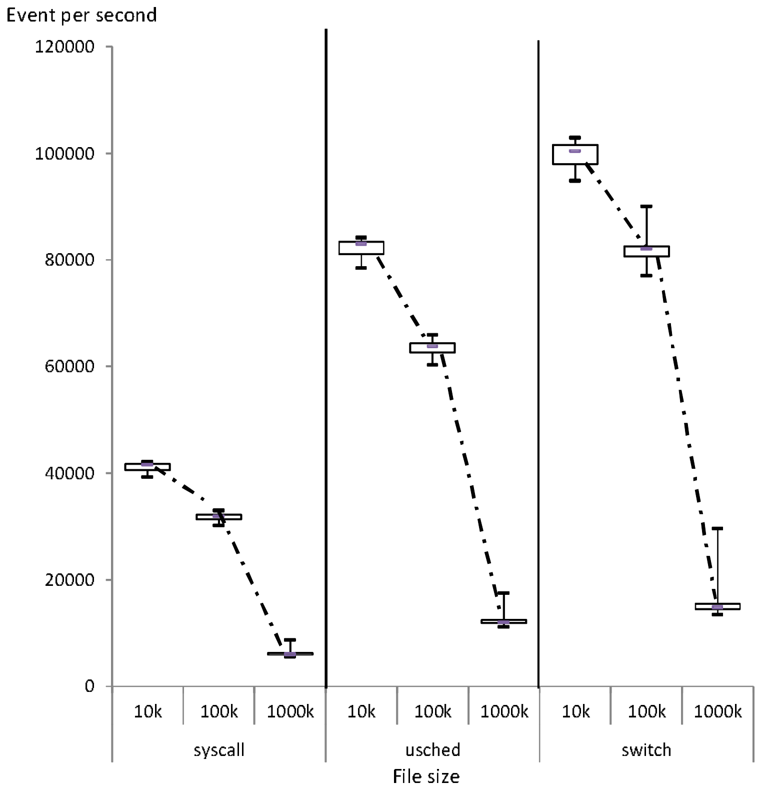


(b) 2vcpus

Figure 46: Web-static CPU utilization

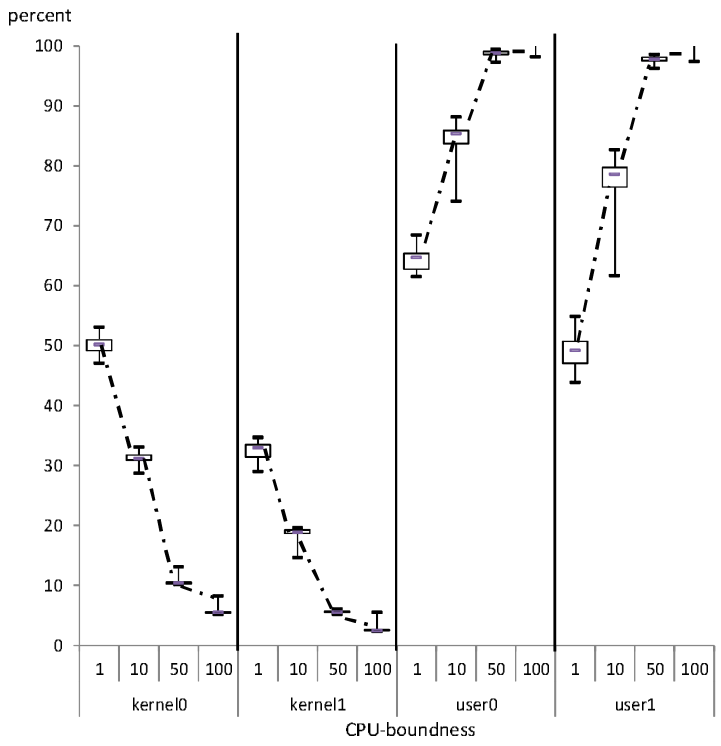


(a) 4vcpus

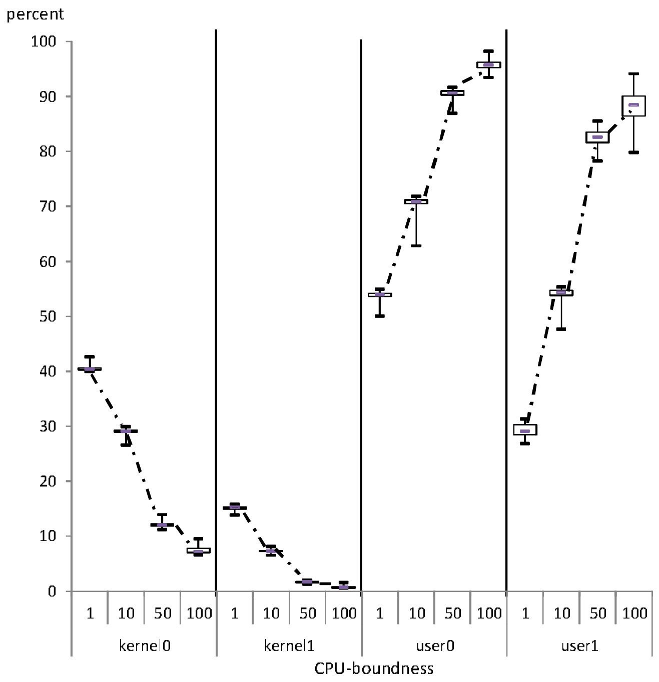


(b) 2vcpus

Figure 47: Web-static stats



(a) 4vcpus



(b) 2vcpus

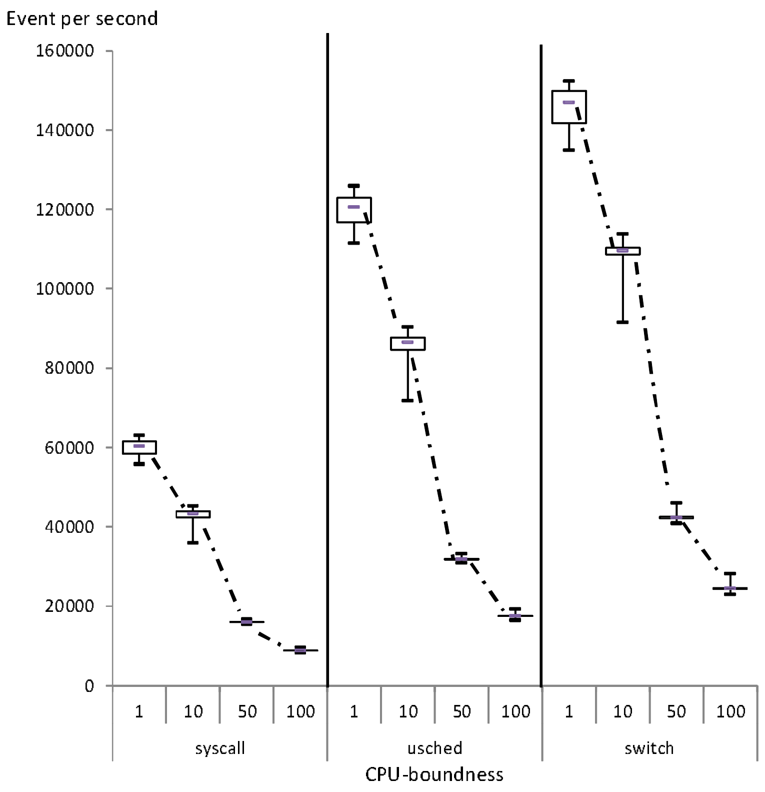
Figure 48: Web-php CPU utilization

while the kernel cores are less active. Even with very low cpu-boundness values (cpu-boundness=1), more user space activity was required than the kernel space activity. Each request to the web server invokes dynamic language processing to generate the web page. This shows the heaviness of dynamic page processing in the web server. As more user cores are used heavily, the system call rates and microscheduling rates drop as in Figure 49.

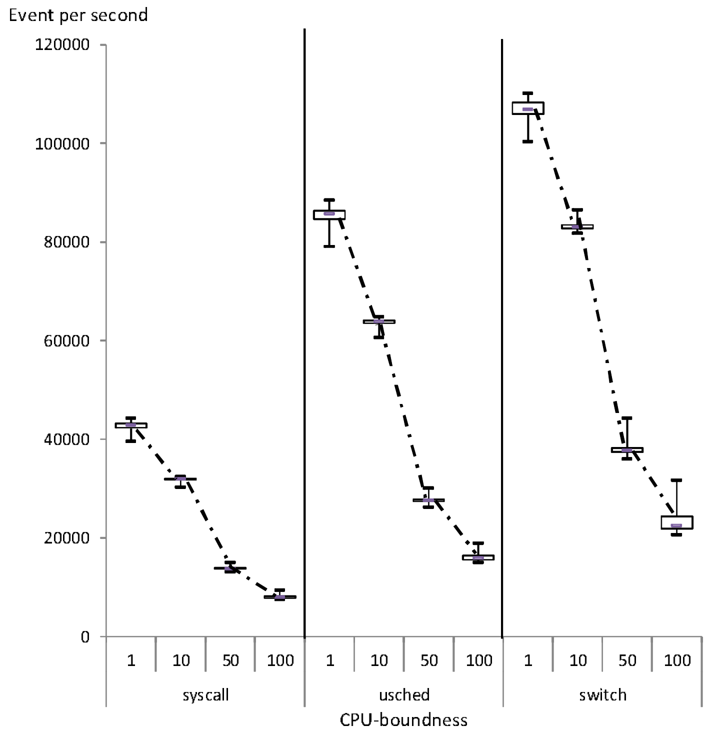
Figure 50 shows the relative performance compared to that obtained when using the regular Xen credit scheduler. X axis is both Web-static and Web-php runs while Y axis is normalized performance based on the number of requests per seconds measured at the client side. “disabled” is the regular Xen credit scheduler while “enabled” is the region scheduler. In case of Web-php (cpu=1,10,50,100) the performance was only half of that of credit scheduler. This is because the kernel cores are idle while user cores are fully used. See the cpu=10,50,100 in Figure 48 where the kernel cores go idle while the user cores are fully utilized and this effectively halves the performance experienced at the client side, as seen in Figure 50(a). This is an example that strict separation of memory regions onto caches may cause unbalanced use of CPU cores. In this case the user space activities are dominant, thus only user cores are active while kernel cores are idle.

The performance impact comes from two major factors. The first factor is CPU utilization issue due to the strict memory-centric approach. As discussed in the previous section 3.4, such unbalanced use of CPU cores can be mitigated when there are ample CPU core resources, compared to number of VCPUs. The relative performance increases at cpu=10, 50, 100 in Figure 50(b) demonstrates that. The Figure 50(b) shows the case of 2 VCPUs rather than (full) 4 VCPUs. This means that the memory-centric scheduler would work better if more CPU cores are provided, compared to the number of VCPUs.

In case of Web-static size=100k,1000k, it shows a bit better performance than

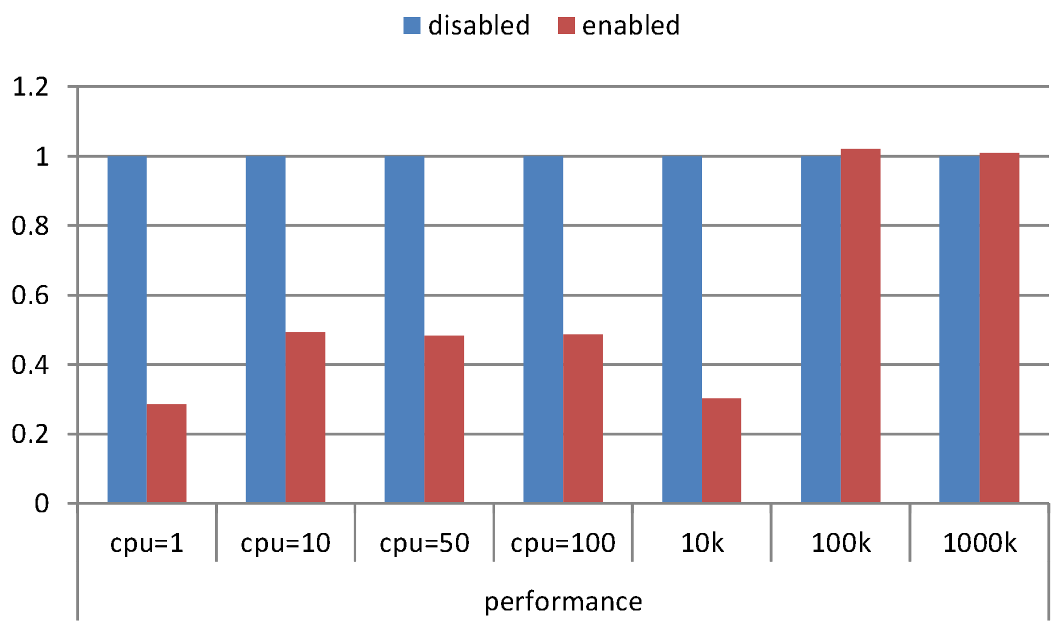


(a) 4vcpus

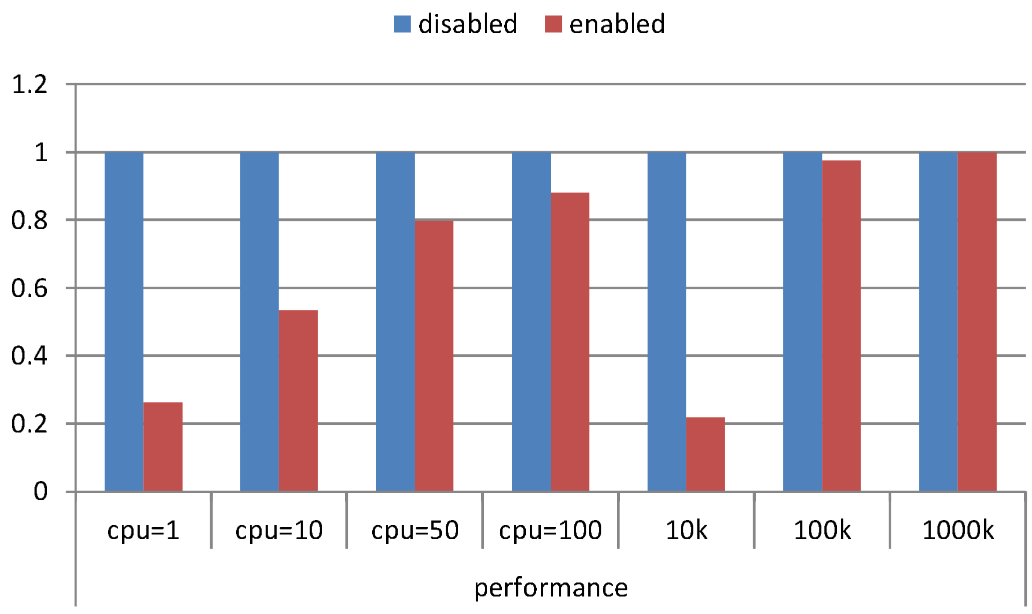


(b) 2vcpus

Figure 49: Web-php stats



(a) 4vcpus



(b) 2vcpus

Figure 50: Performance impact

the Xen credit scheduler. See the Figure 46 where we can find that CPU utilization is quite low for those runs. This indicates that the strict memory-centric scheduler would work better with low CPU utilization environment.

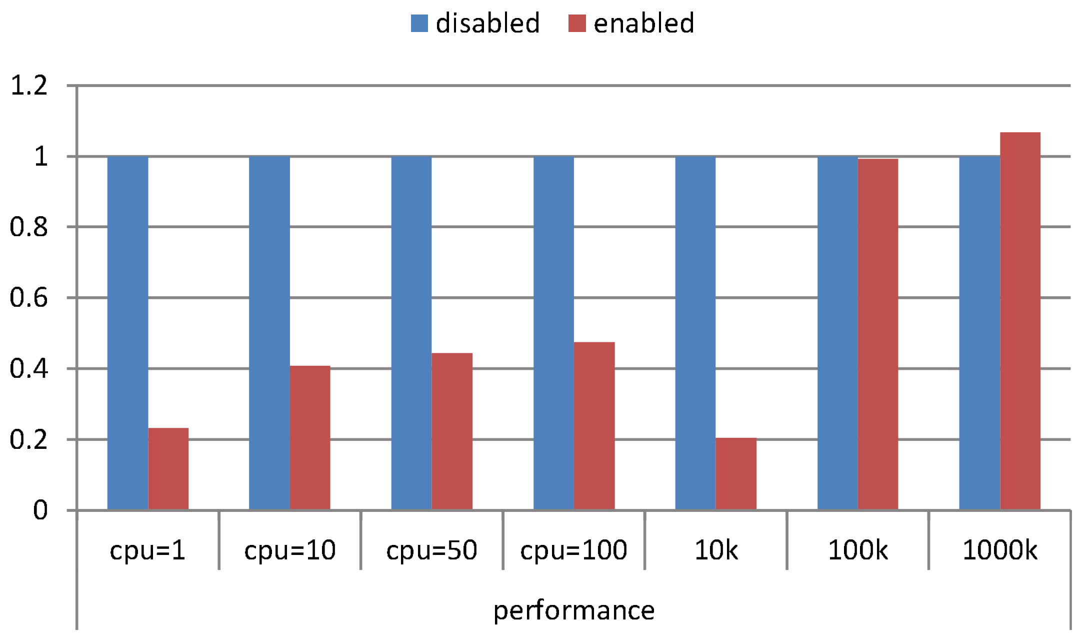
Also note that, at cpu=1 and filesize=10k,100k,1000k, the relative performance is similar for both 4 and 2 VCPUs, because CPUs are not fully utilized, so there are spare CPU resources.

The second factor is the microscheduling overhead. Assuming that using 2 VCPUs eliminates lower CPU utilization issue, the performance impact would be due to the microscheduling overhead. At cpu=1,10,50,100, Figure 50(b) shows microscheduling overhead in proportion to the microscheduling rate in Figure 49(b).

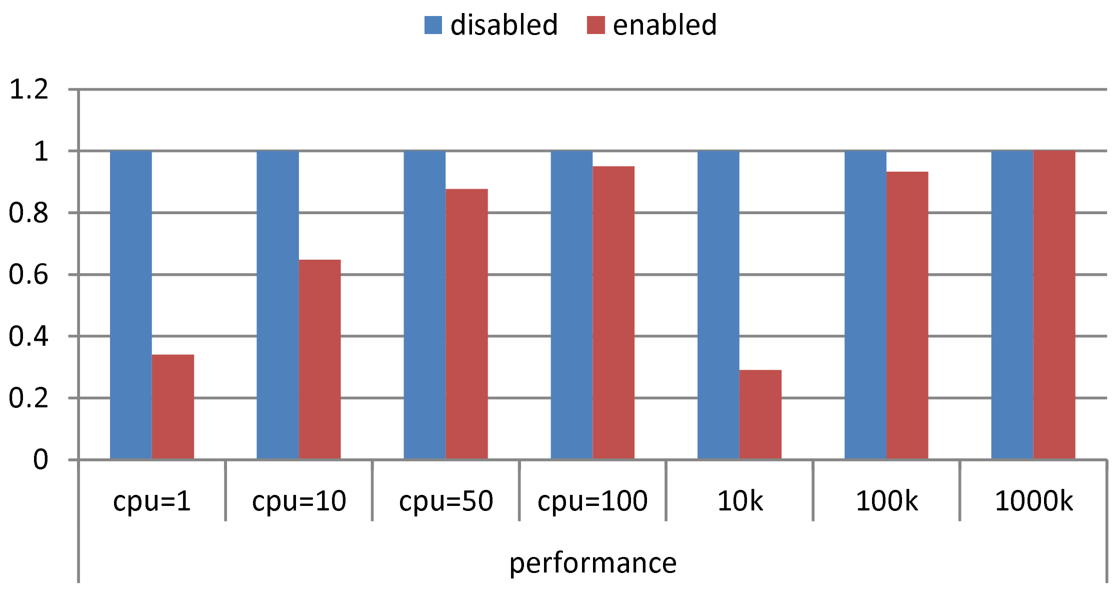
3.5.2 File system cores

Other than just kernel-user space, region scheduler can map a certain subsystem using guest-defined regions. In this subsection, two cores are declared as file system cores and the other two cores are declared as "user and other" cores. This demonstrates how certain subsystem can be mapped to certain cache. In general, by mapping file system to certain cores, the microscheduling rate and the system call rate dropped significantly. The microscheduling rate dropped to less than 40000 in Figure 52 and this is 1/2 or 1/3 when it's compared to kernel core case (Figure 47 and 49). This is as expected since only file system activities are microscheduled, rather than whole kernel space. Also we can see that the microscheduling rate is now dropped below the level of system call, unlike kernel core cases where microscheduling rate was much higher than system call rate. This is because only file system activities are microscheduled. This reduced microscheduling rate improves performance as below.

Figure 51 shows the performance impact for the file system core. In general, the performance shows the same trend as before, but it shows some improvements. In the case of 4 VCPUs, it shows a similar performance with kernel cores (Compare Figure



(a) 4vcpus



(b) 2vcpus

Figure 51: Performance impact for file system core

51(a) and Figure 50(a)). Meanwhile, in case of 2 VCPUs, it shows better results than kernel cores up to 10% (Compare Figure 51(b) and Figure 50(b)). This is because of the significantly reduced microscheduling rate.

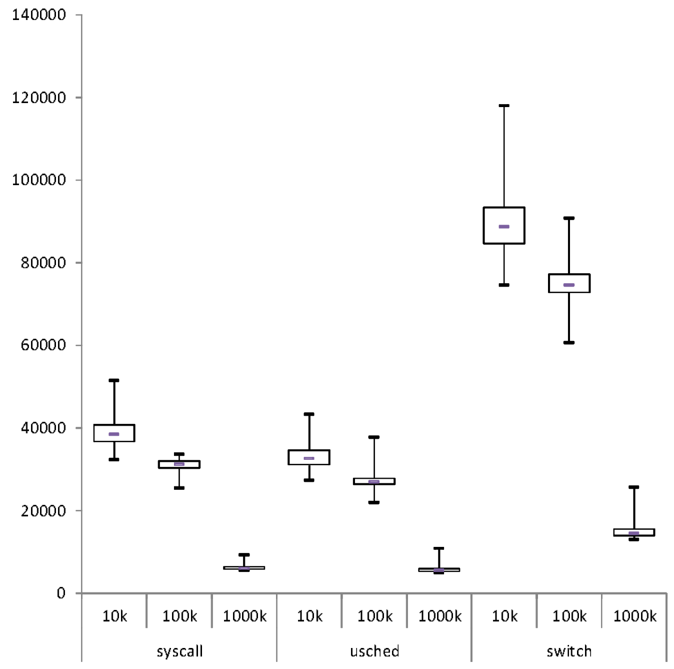
Similarly, Figure 53 shows the case of network core where networking is mapped to one cache, and Figure 54 shows the file system and network cores (fsnet) where networking and file system both are mapped to one cache. Although the performance is better than that of file system core case, lower CPU utilization issue still blocks better performance than Xen scheduler.

Figure 55 shows the number of API calls to set up guest-defined regions. X axis is time in second and Y axis is the number of API calls. This basically shows that the API call overhead is negligible. Once the slap allocator in the guest establishes page-to-region associations, it rarely changes the association.

In sum, this section shows the cost of rigid (or strict) partitioning of each space onto caches. It causes a high rate of microscheduling and substantial overhead. Such overhead can be significantly reduced by relaxing the separation of two spaces – use space and kernel space. One approach is to simply globalize some regions or pages. Next section discusses about this.

3.5.3 Discussion

In this section, we demonstrated the strict memory-centric scheduler approach (the strict partitioning of each space onto caches) and discussed about its potentials and limitations. We did not introduce globalization yet, thus it is called strict memory-centric approach and we observed how such strictness introduces unbalances or high overhead. Therefore, we would like to conclude as follows. In general, the strict memory-centric approach has an issue of unbalanced core usage. However, under certain conditions such as lower VCPU-to-PCPU ratio and low CPU utilization, the issue of unbalanced core usage is mitigated and it may see some performance benefits



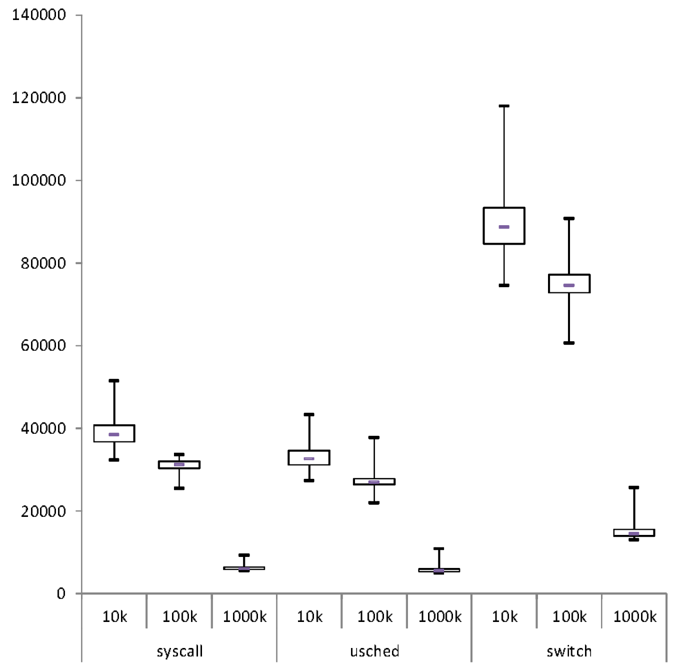
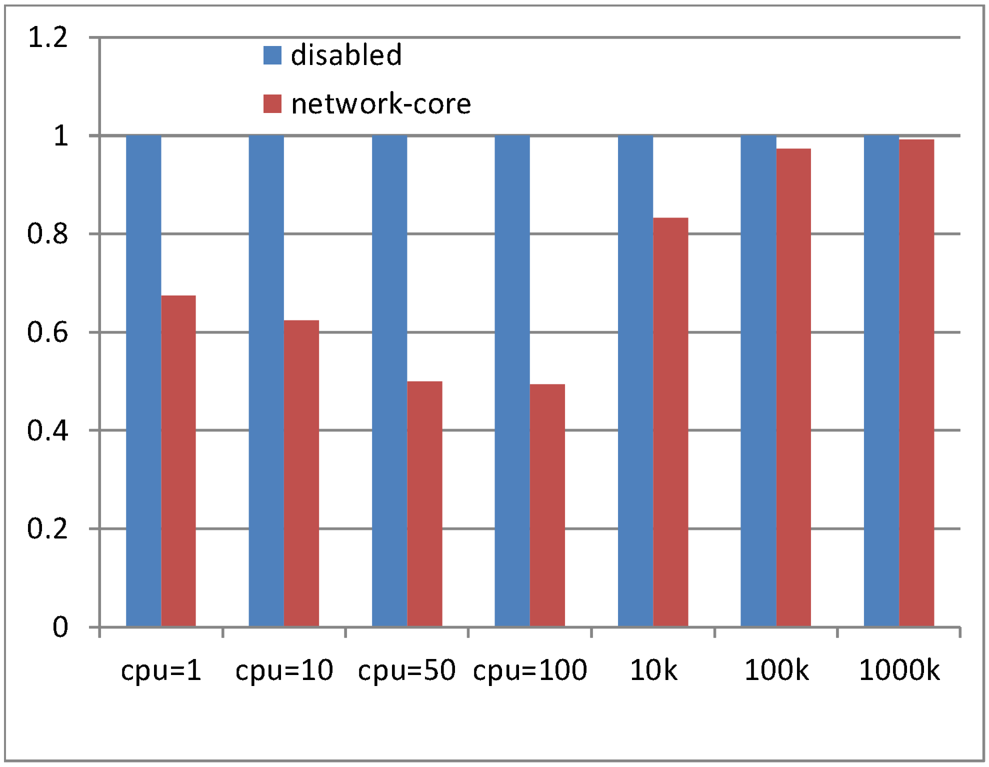
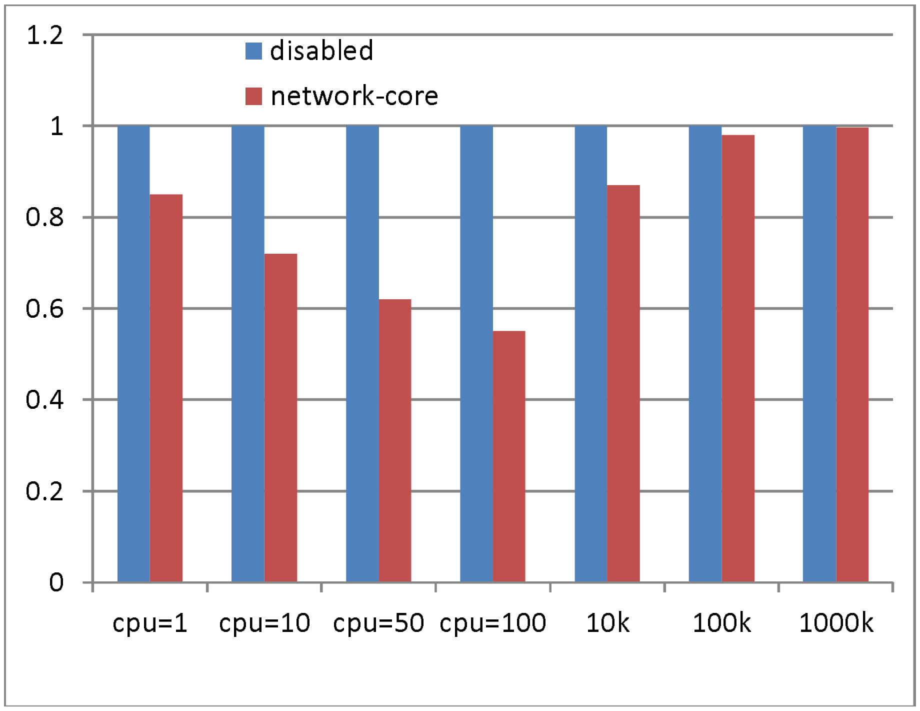


Figure 52: Reduced usched (file system core)

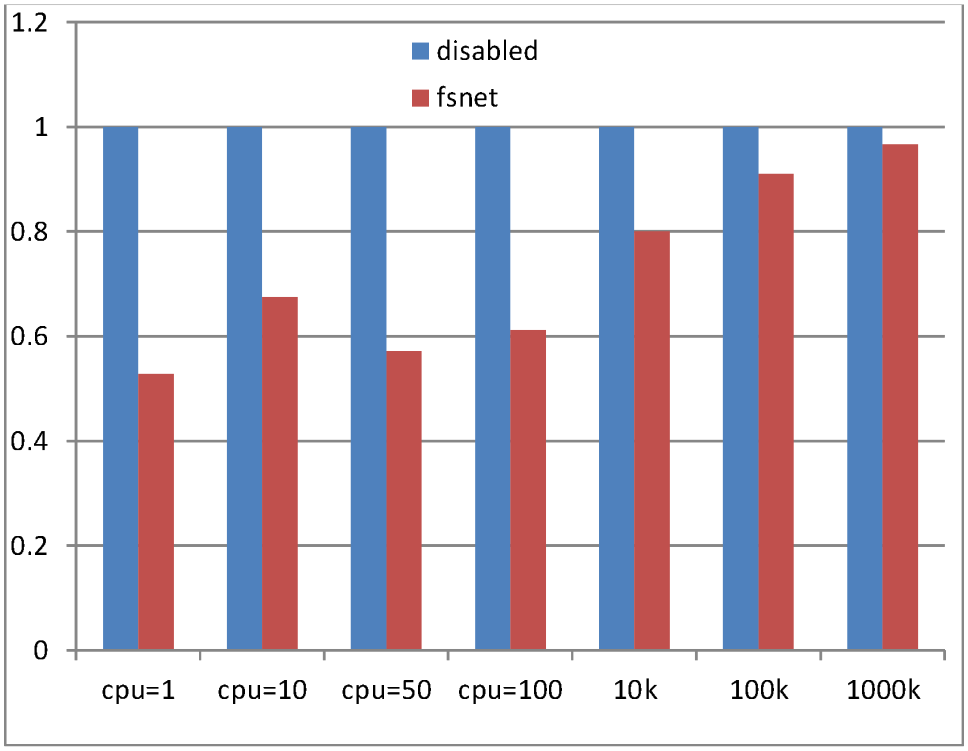


(a) 4vcpus

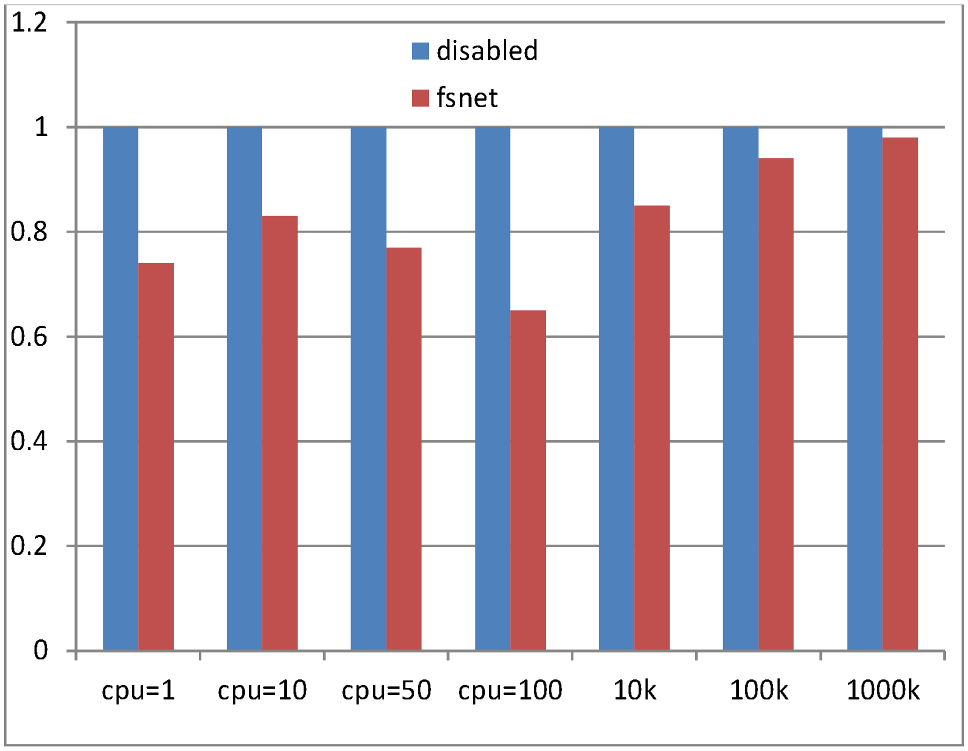


(b) 2vcpus

Figure 53: The case of network core

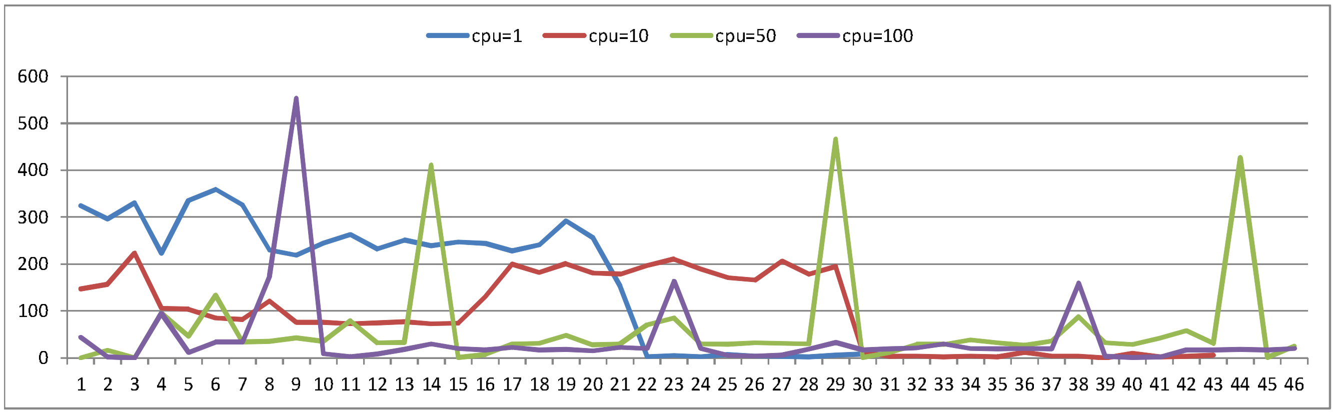


(a) 4vcpus

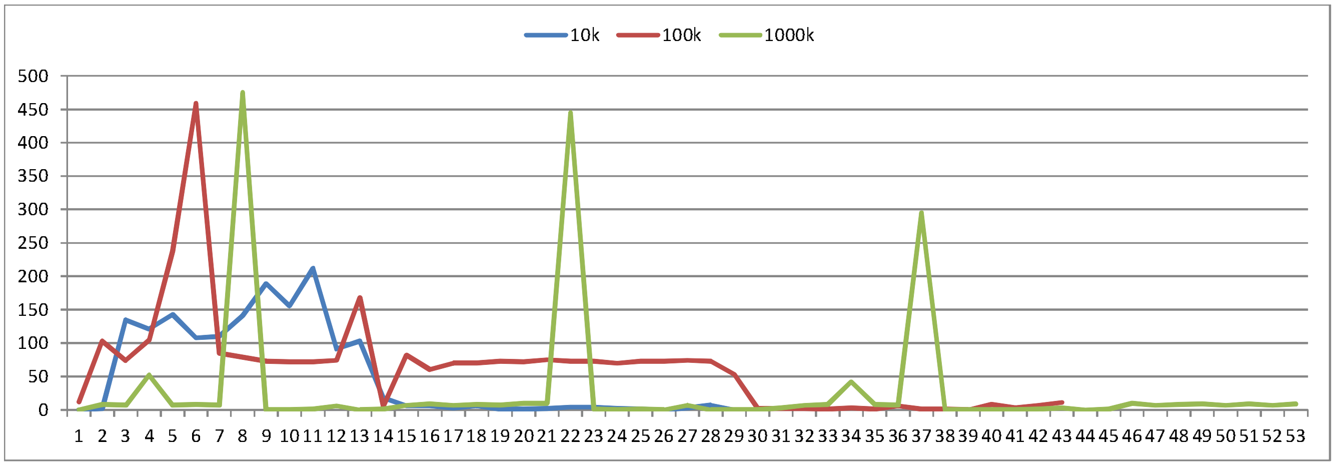


(b) 2vcpus

Figure 54: The case of file system and network (fsnet) core



(a) web-php



(b) web-static

Figure 55: API call overhead: X axis is time in second, Y axis is the number of API calls

due to the cache effectiveness. Also the strict memory-centric approach has an issue of high microscheduling. It may cause high microscheduling overhead so it is necessary to relax the strictness to control the high microscheduling rate.

3.6 Globalization

Section 3.5 details a case of single VM, simple web server. It not only shows the possibility of kernel disaggregation by functionality, but also shows some limitations of this approach. The major issue is that it may cause a high rate of microscheduling due to the strict region-to-cache mapping. To remedy this, we introduce globalization that permits some global mappings of regions. Those global regions are accessible through all PCPUs. This dramatically reduces the microscheduling rate with consequently reduced overheads.

3.6.1 VM Consolidation

Cloud environments routinly see workloads consolidated onto some small number of physical machines [109]. For example, application tiers may be placed onto machines running the database tier, to deal with abrupt spikes in load [108]. This helps scaling, but can cause interference as workloads share underlying machine resources. In response, one approach [108] simply pins each workload onto some specific cores, but this does not offer good isolation (workloads share other machine resources, like caches and memory bandwidth) and leads to under-utilization of machine hardware (pinning means that workloads cannot move even when other resources are currently idle).

Region scheduling benefits from high levels of consolidation, as there are more VCPUs than PCPUs, which makes it likely for some VCPU to be schedulable, thus avoiding the potential performance degradation due to core idleness seen in the experiments above. We next explore this topic experimentally.

3.7 Evaluation

In this evaluation, four-socket Intel Westmere server machine is used, so we consider four caches that each is 24MB. The consolidated workloads studied in this section run front tier applications like the webserver, back tier applications like mysqld database server, and the memcached caching system as a simple example of application tier functionality.

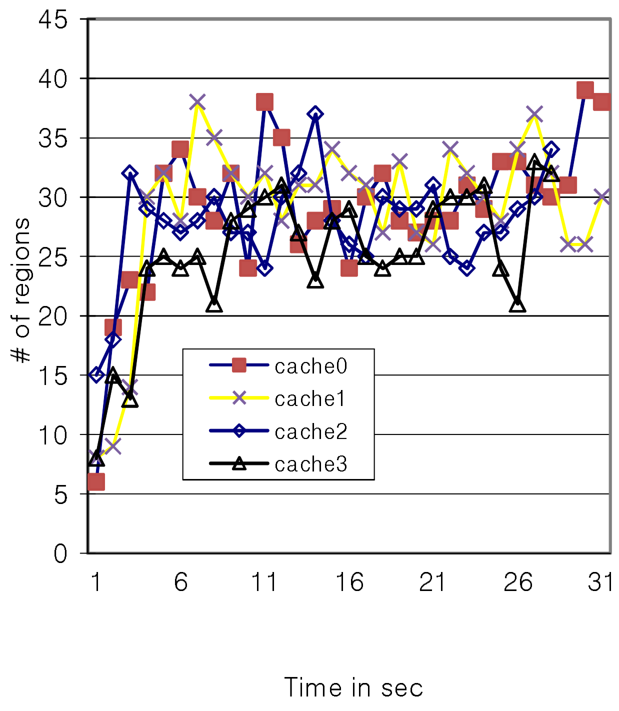
3.7.1 Memcached

Memcached is a memory-caching system used to speedup dynamic pages on websites. It caches data and objects in DRAM, thereby reducing database accesses. We use three well-known three PHP codes that generate dynamic web pages MediaWiki [115], WordPress [116], and XpressEngine [117]. MediaWiki is PHP code originally written for Wikipedia. WordPress and XpressEngine are content management systems (CMS) used for websites, blogging, etc. Memcached can be used with these to speed up their dynamic web page generation.

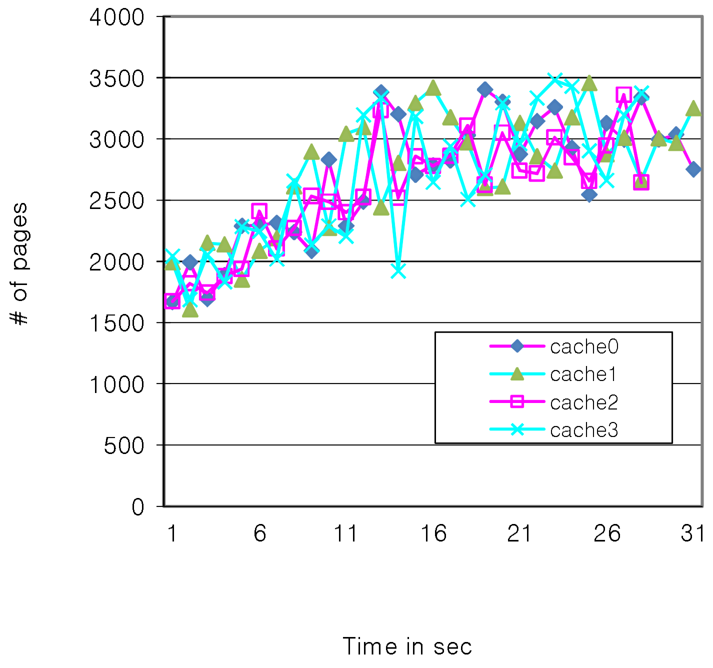
Name	**Description**	**Improvement**
MediaWiki	Wiki package in PHP	1.8%
WordPress	Web site or blogs	4.6%
XpressEngine	Web site or blogs	-2.8%

Table 4: Three PHP codes for dynamic page generation

Figure 56 and Table 4 shows single VM case. Although two positive impacts of 1.8% and 4.6% were observed, also there was a negative impact of -2.8%. For this run, on average, roughly 120 regions (4*30 in Figure 56(a)) are mapped to each cache (cache0, cache1, cache2, cache3 in Figure 56). Considering the number of pages that is mapped to each cache in Figure 56B, 3000pages, about 12MB of memory is mapped to each cache. Although no performance benefits were observed for this single VM case, the region scheduler is mapping each memory regions to the caches.



(a) Number of regions



(b) Number of pages

Figure 56: Single VM, WordPress+memcached

3.7.2 VM Consolidation

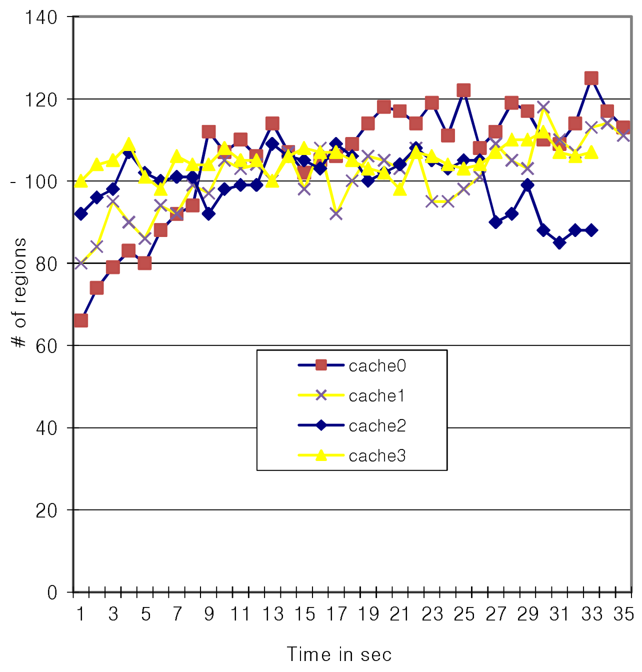
To emulate higher levels of consolidation, we add two additional VMs, AsteriskVM and SPECVM. AsteriskVM runs Asterisk, which is a popular IP-telephony workload, or a SW telephone private branch exchange (PBX). SPECVM runs the SPEC2k6 CPU benchmarks.

Name	**Description**
Apache2+Mysqld	Web server + DB
Asterisk	IP telephony
SPEC2k6	SPEC CPU benchmark

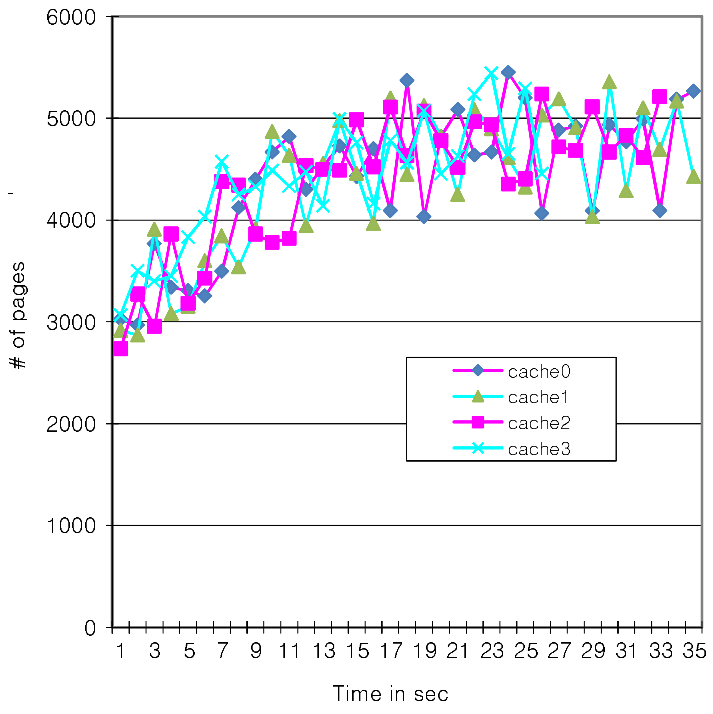
Table 5: VM Consolidation

As is apparent from the Figure 58, there is some positive impact on performance from the higher levels of consolidation used in these experiments. Also 2 to 6% improvements are observed for Apache2 web server on the client side. As more VMs are added, there are more VCPUs than PCPUs, which leads to more regions being detected, as shown in Figure 57. On average, 400 regions are formed and they are mapped to each cache, as per the cache balancing methods part of region scheduling. As a result, performance is increased by up to 5% for SPEC VM, and the response time for Asterisk VM is improved by up to 10%.

Nonetheless, with the applications and kernel functionality evaluated to date, we see only modest gains from disaggregating kernel functionality by memory regions. We hypothesize several causes for this. First, micro-scheduling overheads remain high for the experimental runs performed in this chapter. Reasons include a potential lack of modularity of kernel functions, i.e., given the complex nature of kernel code, it is difficult to confine say, the file system functions to interact only with each other and not with other kernel functions. The latter, however, will result in microscheduling overheads. This may lead to short runs and frequent microscheduling. Second, typical kernel activity such as file I/O is short, which implies that they may not run

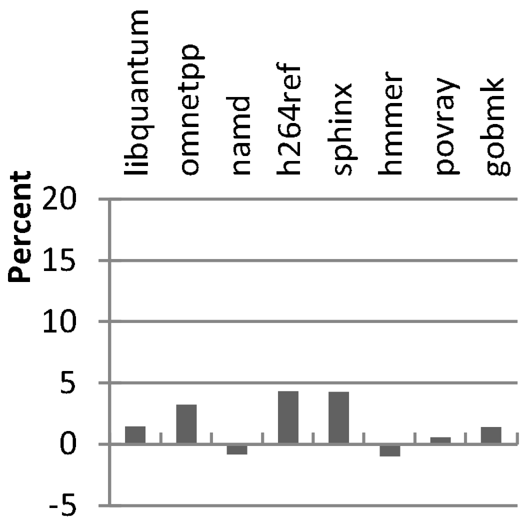


(a) Number of regions

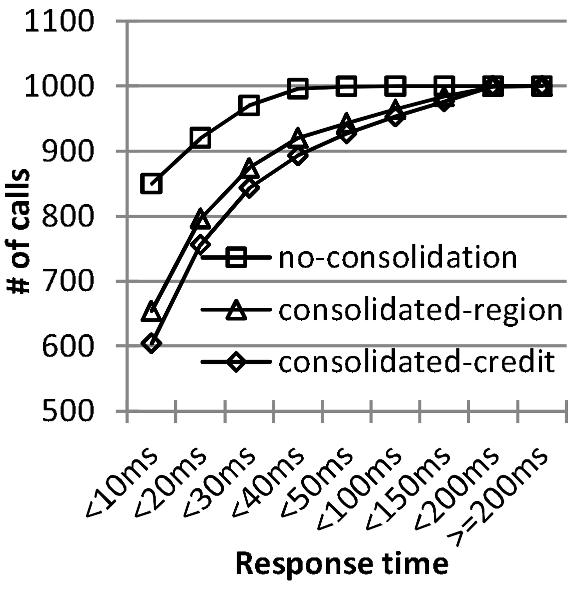


(b) Number of pages

Figure 57: VM consolidation



(a) SPEC VM



(b) Asterisk VM

Figure 58: Performance impacts

sufficiently long to amortize the overheads of (i) microscheduling when the VCPU start the operation, followed by (ii) microscheduling when returning to the previous activity. Up to roughly 10000 microscheduling per a second is observed and this was quite high rate. Third, the globalization strategy eventually grows globalized memory portion and this is effectively reverting region scheduler to the Xen credit scheduler. Although this would reduce microscheduling rate, at the same time we lose the benefits of the regioning scheduler.

Therefore, in overall, applying region scheduling to the kernel level seems limited in its usefulness in spite of its novel approach of specializing cores for each functionality. This is in part because of the nature of kernel core that it is shared across all processes . For example, the file system is used by all processes and hence good spatial locality on files gets lost from the kernel's perspective.

3.8 Related Work

3.9 Conclusions And Future Work

Recent work such as FlexSC is close to our work in that it also creates the notion of user cores and kernel cores. FlexSC [106] replaces the traditional synchronous system call with an asynchronous exceptionless system call mechanism. In the asynchronous system call mechanism, the user thread uses simple RPC-like mechanism to request OS services on other core. The user thread simply marks a flag then switch to next user thread in the threads pool. In this way, the user core runs only user code while the other cores runs only kernel code. This approach can be viewed as an user cores and kernel cores. It also briefly mentions the interesting scheduling issues regarding user core and kernel core discussed in this chapter. However, FlexSC requires new threading pacakages in the user space and it handles whole kernel space rather than its subsystem because it intercepts the system call. Compared to our work, it has benefits of new user level threading pacakage and simple message passing. It uses a new user level library based on simple message passing, instead of microscheduling. In our work, we targets virtualized environment with transparent support for guests. Therefore, in the region scheduling, there is no need to modify the applications, and the hypervisor deals with disaggregation, entirely through its use of the scheduling and region layers. This allows fine-grained core specialization such as file system cores and network cores.

3.9.1 Discussion

This work demonstrates kernel service disaggregation by region scheduling, transparent to guests. The VCPUs are microscheduled to the cores where the desired functionality is mapped to. The functionality such as file services is mapped to each cache unit because the cache is what makes big difference between cores. Because the region scheduler manages the last level cache and it already maps the memory region to the caches, it could be leveraged naturally to implement mapping the functionality to the cores. This may be viewed as an specializing core groups to a specific functonality. We have demonstrated the kernel core where all kernel services are mapped, and the file system core where the file system is mapped. Similarly, the network cores are demonstrated. Then, the load balancer conflict is identified and analyzed, so the globalization strategy is introduced to overcome such limitation and it was useful to control microscheduling overhead. Also we considered VM-consolidated environments such as cloud environments as an oversubscribed systems. However, in general, only modest gains were achieved in spite of successful mapping of kernel functionality to the caches. The approach is good in that the caches are more efficiently used by the workloads, but in case of kernel space, they seem to have small footprint and often shared across processes. This likely makes this approach less attractive for kernel spaces.

However, if we take a look at other side of this work other than the performance improvements, this work may have a potential impact on the heterogeneous systems. The region scheduler is used to functionally disaggregate kernel services to certain CPUs and, in general, any heterogeneous functions can be mapped to different CPU cores. Furthermore, in theory, it is possible for each CPU to have different instruction set architecture (ISA). In this case, code can be a mixture of heterogeneous ISAs and each type of instructions would be mapped to the corresponding CPUs. By using the memory region abstraction, each code do not have to be annotated which ISA it is, but they can form a code region which can be automatically microscheduled to the correct cores.

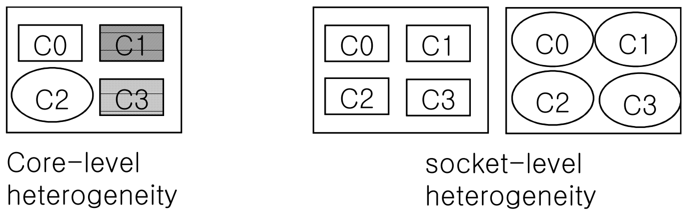


Figure 59: Platforms consisting of heterogeneous resources

Modern computing platforms are heterogeneous in several of their resources. Heterogeneous processors, consisting of CPU cores that are different in their performance/power capabilities, have been proposed as an energy-efficient alternative to homogeneous configurations [50, 56, 67]. This form of performance heterogeneity can exist at many levels: cores within a socket or across sockets, as shown in Figure 59. There are several commercial implementations of such heterogeneous CPU architectures [45, 55, 61, 78]. Several studies have shown that different processor architectures are suited for different applications. For example, prior work has discussed the utility of low-powered cores for the design of datacenters [41, 62] as well as the need for high-performance brawny cores [43, 68]. Various scheduling methods for heterogeneous cores have also been investigated [58, 66, 70, 85]. Use of the DVFS (dynamic

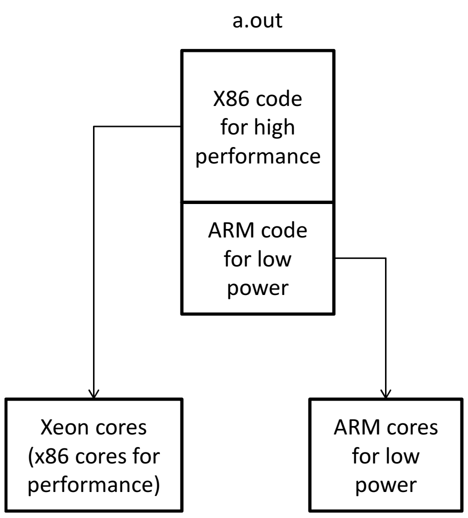

a.out
X86 code for high performance
ARM code for low power
Xeon cores (x86 cores for performance)
ARM cores for low power

Figure 60: Runs across heterogeneous ISA architecture

voltage and frequency scaling) [81] also creates another kind of heterogeneity. Moreover, processors can be heterogeneous in the levels of performance offered [50, 56, 67], like the big/little cores commonly found in today's client systems [45, 55, 78]. Also in multi-socket system, I/O configuration is one other factor that creates heterogeneity. For example, some cores or sockets can be near the I/O bus than others [122].

With the likely increased use of heterogeneous ISA cores, or functionally asymmetric cores, one of the major obstacles for effective use of such heterogeneous platforms is their programmability. The region abstraction can be used to make it easier to program future heterogeneous systems. The simple idea, shown in Figure 60, is as follows. We assume a program to be either already able to run on some certain processor and/or for it to be compilsed into some multiple-ISA binary code (another option is to use fault and migrate when ISAs are only partly heterogeneous [124]. We then use the region scheduling framework to match code sections to corresponding ISA cores. We next consider 'memory heterogeneity' in future multicore systems, again using region scheduling to deal with its effects.

CHAPTER IV

MEMORY HETEROGENEITY

4.1 Introduction

We next present a software-controlled technique for managing the heterogeneous memory resources of next generation multicore platforms. With new memory technologies like PCM (Phase change memory) , NVRAM emerges, there is a need for heterogeneity-aware system software capable to exploiting and dealing with memory heterogeneity [121]. We illustrate the ability of the region scheduling approac to cope with such heterogeneity by considering 3D-stacked DRAM, which provides low-latency accesses to relatively small amounts of DRAM located on a chip. We implement our solutions in the Xen hypervisor for virtualized environments, and this chapter demonstrates how the hypervisor can handle such heterogeneous memory configurations. Experimental evaluations emulate differences in memory access latencies by using NUMA nodes on a multi-socket Westmere machine, slowing down accesses to these nodes by throttling the memory controller and reducing the memory in one of these nodes to the smaller memory capacities expected for future 3D DRAM configurations. In this case, the main question is which memory pages should be in the valuable 3D-stack DRAM. Because 3D-stack DRAM is limited in its size, compared to the traditional off-chip DRAM, it is critical to intelligently choose such pages. Since the region framework has the capability to track working sets, we utilize the region framework to implement policies for this emulated heterogeneous memory.

Because this is transparent to guest VMs, it is also transparent to the applications, which shows that the region framework unburdens programmers, hiding from them the details of heterogeneous memory management. Further, VM implementations are not affected, which is important because it enables easy VM migration among heterogeneous nodes. This would not be the case if the VM or application were programmed or optimized assuming certain heterogeneous memory configurations, a case in point being the current separation of (discrete) accelerator from host memory. Benefits will be particularly apparent in public clouds like Amazon EC2 [38] and Google's Compute Engines [54], as for such VM deployments, the region framework hides the underlying complexity associated with memory heterogenity from guest VMs. More specifically, with region scheduling, guest VM memory usage can be adapted for the memory configurations seen on underlying hardware, at hypervisor level and without the need to involve guest VMs.

The region framework does online detection of hot memory pages and transparent page migration. It offers a simple abstraction of homogeneous scalable virtual memory to guest VMs, which is then mapped appropriately to underlying heterogeneous memory. Our experimental evaluation on an emulated heterogeneous memory platform uses workload traces from real-world data, demonstrating the ability to provide high on-demand performance while also reducing resource usage for these workloads.

We note that another advantage of using regions and region scheduling with VMs on heterogeneous memory platforms is improved elasticity in VM resource assignment. Currently, resource allocations for VMs are coarse-grained, with commercial cloud services like Amazon EC2 AutoScale [39] relying on server-level scaling via fixed instance types. Further, these instances can only be rented on the order of several minutes to a full hour, and applications are charged for the entire instance even if it is only partially used [40]. Thus, scaling out is a rather heavy-weight and coarse-grained operation, having high cost implications for the end-user and limiting the frequencies at which these scale-out mechanisms can be used. For example, changing a small VM instance to a large VM instance in Amazon EC2 requires shutting down the current VM, changing its properties, and rebooting the VM – which is a rather slow operation.

Server scaling also does not provide a way to improve the performance of existing resources owned by an application, e.g., moving from a VM instance to another type of instance with different resources requires a VM restart in Amazon EC2. Previous work has highlighted the need for long running instances in datacenters for predictable performance [44]. Customers could implement their VM-internal solutions to this problem, but this is complex. Finally, these mechanisms may require application-level changes to deal with new resources such as load-balancing across instances, etc.

The region framework enables fine-grained memory page allocations across the heterogeneous memory of underlying machines. For example, a VM may want to allocate only 10% of its DRAM memory in 3D-stacked fast DRAM to optimally run its workloads, then incrementally increase or decrease its portion as its loads or requirements change. This enables better pricing strategies for both the customer and the cloud computing providers [40], because it allows frequent adjustments of the resource allocations of applications. Given the competition among cloud providers for cheaper/better services, fine-grained resource management may prove to be a compelling feature of future cloud platforms [37]. Another advantage to this approach is that it can be transparent to the VM and applications, not requiring sophisticated software changes to deal with varying resources. Several techniques have been proposed in the literature to enable fine-grained resource management in clouds [44, 82, 80, 86]. However, each of these focus on a specific approach, without a unifying mechanism to enable their adoption.

The presence of heterogeneous memory brings significant management challenges since application performance with respect to memory is governed not by the total amount of fast vs. slow memory allocated to an application, but instead, by the fast vs. slow memory speeds experienced by an application's current memory footprint [94]. To address these issues, the region framework scans the access-bit history for each VM, obtained by periodically scanning the access-bits available in page tables. The history is used to detect a guest's 'hot' memory pages, i.e., the current memory footprints of the running applications. In addition, guest page tables are mirrored in the hypervisor, allowing it to manipulate guest page mappings in a guest-transparent manner, thus making possible page migrations (between slower vs. faster memories) by simply changing mappings in these mirror page tables, without guest involvement.

The region framework is implemented in the Xen hypervisor. In order to evaluate its utility and overheads with actual applications and workloads, not relying on architectural simulators, memory controller throttling is used to emulate memory subsystem heterogeneity. With workloads that use traces from Google cluster data [59], experimental evaluations show that by exploiting heterogeneity in an unobtrusive way, the region framework makes it possible to achieve on-demand performance boosts and cost savings for cloud applications with diverse resource requirements. Specifically, the memory scaling mechanisms reduces, on average, 30% memory resource usage and thus, the memory costs for these workloads.

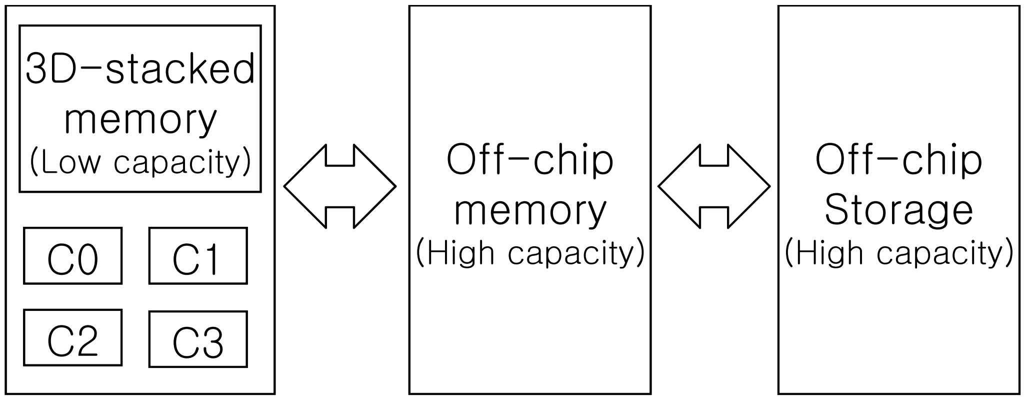


Figure 61: Hetero memory

The introduction of new memory technologies such as die-stacked 3D memories, NVRAM , in addition to traditional DRAM, can result into a hierarchy of heterogeneous memory organization shown in Figure 61. 3D stacked memories can provide lower latency and higher bandwidth, in comparison to traditional off-chip memories [96]. However, the capacity of such memories is likely to be limited to only a few hundreds of megabytes [97]. Thus, a combination of both fast on-chip memory with additional slower off-chip memory is needed for higher capacity and expansion capabilities, specially for high-end enterprise machines. Further, addition of disaggregated memory or persistent memory technologies can also result in memory heterogeneity [91, 101, 92, 95]. Other types of memory includes persistent memory, flash memory.

4.1.1 Heterogeneous memory

The recently proposed 3D die-stacked memory [96] not only provides lower access latency and higher bandwidths but also provides lower power in comparison to traditional off-chip memory. Because it is stacked on-chip, it is likely to be constrained in its capacity. The capacity is projected to range only up to a few hundreds of megabytes [97] and this suggests a usage model where such on-chip memory is combined with the off-chip memory providing both higher capacity and low latency. Figure 61 depicts such a heterogeneous memory subsystem, consisting of slow off-chip memory and fast on-chip memory.

The 3D stacked DRAM on chip can be used in two ways. First is to use it as an hardware-managed cache. This has an advantage of being able to quickly react to changing memory access patterns. Also it provides a transparent way to incorporate such new memory architecture in a way that supports legecy applications. However, it has some drawbacks and challenges. It would result in high overhead for managing the tags of such large sized caches and it would require an extended coherency support. These all consumes more power. Also diminishing returns are expected due to the big size of the cache.

The second is to use it as an software-managed memory. In this chapter we explore options that an operating system or hypervisor can take. Such system software can use its high-level information about the application's behavior to manage such heterogeneous memory resources. Specifically we will take a look at 3D stacked DRAM that is explicitly exposed as a system-visible memory to the hypervisor. Then the feasibility of software-based memory management is evaluated for such heterogeneous memory platforms.

Specifically, we assume the virtualized environment as the use of virtualization in server systems increases. Then, several challenges for managing such heterogeneous memory resources are identified. First, the CPU provides only limited information about the memory access patterns of the guest VMs. In case of x86, it only provides one-bit information called access bit in the page tables. Thus, this requires efficient methods to detect which pages are critical for a guest's performance based on such limited information from hardware. Second, to retain transparency to the guest OSes, the hypervisor should handle the page tables separately from that of guest VMs. Implementing the management scheme transparently may be challenging because the page tables are owned by the guest in paravirtualized environment. This also would make it difficult to migrate any of its pages between memories without guest involved. In case of the full virtualization where the hardware virtualization support is used, additional layer of page table can be used, but even in this case, such mappings should be handled properly. Also, TLB management across cores should be handled properly.

In this chapter, we present techniques to address these two points above. (1) The hypervisor is enhanced to build an access-bit history for each virtual machine, periodically scanning the page tables for collect access-bits. This 'A-bit history' is then used to identify 'hot' pages and calculate the working set size. (2) This scanning page table may incur significant overhead; we introduce many optimizations such as additional data structures for quick accessing page table entries. (3) Also we carefully design the scanning algorithm so the virtual time of guest VMs are used to determine scan intervals. This contributes to the accurate accounting. (4) To enable transparent page migrations, the hypervisor is extended to mirror the guest page tables. The hypervisor uses the mirrored page table not to disturb guest while it migrates pages. This enables transparent page migration.

Mechanisms such as page access tracking, hot page detection, and mirroring page table are all implemented in the Xen open-source hypervisor. So it enables experimental evaluation of overheads in realistic server platforms. Then, a multi-socket Intel Westmere platform is used to emulate such future memory heterogeneity. One of the machine's memory controllers is throttled resulting in the presence of both 'fast' and 'slow' memory. One regular DRAM node acts as 'fast' memory and the other throttled DRAM node acts as an 'slow' memory in the system. Therefore, such experimental results on this machine and memory configuration characterizes the memory behavior of the standard server workloads in terms of their working set sizes and the performance impact of memory heterogeneity. We evaluate the page migration mechanism by micro-benchmarks, to show the feasibility of software management for future heterogeneous memory systems.

In summary, this chapter's technical contributions include:

- A hypervisor-level mechanism to detect guest memory access patterns using access bit information.
- Transparency support for managing heterogeneous memory for virtual machines, implemented by the hypervisor.
- An evaluation of the sensitivity of several server workloads to the performance of heterogeneous memory subsystems.

In the remainder of this chapter, we first describe the mechanisms used for tracking guest activity and policies for stacked memory allocation in Section 4.2. Section 4.3

describes our evaluation methodology, with experimental results presented in Section 4.3.3. Finally, related work and conclusions are described in Section 4.5 and 4.6 respectively.

4.2 Heterogeneous Memory Management

Concerning the memory subsystem, the region framework provides the interface of a performance-scalable memory over heterogeneous components by changing a VM's allocation in different memories, i.e., use of fast memory for high-performance E-states and slow memory for slower E-states. This section describes elasticity management for heterogeneous memories involving fast die-stacked memory and slow off-chip DRAMs. Since the die-stacked memory is small in capacity in comparison to the off-chip DRAM, a subset of pages from application's memory need to be chosen to be placed in the stacked-DRAM. For this purpose, it is important to detect and manage application's 'hot' pages that are critical to its performance which in turn requires efficient way of memory access tracking.

4.2.1 Memory Access Tracking

Modern processors provide only limited support from hardware for detecting applications' memory access patterns. On x86, each page table entry has an access bit, which is set by the hardware when the corresponding page is accessed. Software is responsible for clearing/using this bit. We use this single-bit information and build access bit history to determine a VM's memory access pattern. Specifically, we periodically scan and collect the access bits, forming a bitmap called as 'A-bit history' (access-bit history) shown in Figure 62. A 32-bit word and a 100ms time interval is used for scanning implying 3.2 seconds of virtual time corresponding to one word. If the A-bit history has many 1's, i.e. dense A-bit history, it would indicate that the page is hot and frequently accessed. A threshold of 22, obtained experimentally, is used in our work for marking a page as hot. We also adopted many optimizations to minimize page table scanning overheads, which is discussed later in this section.

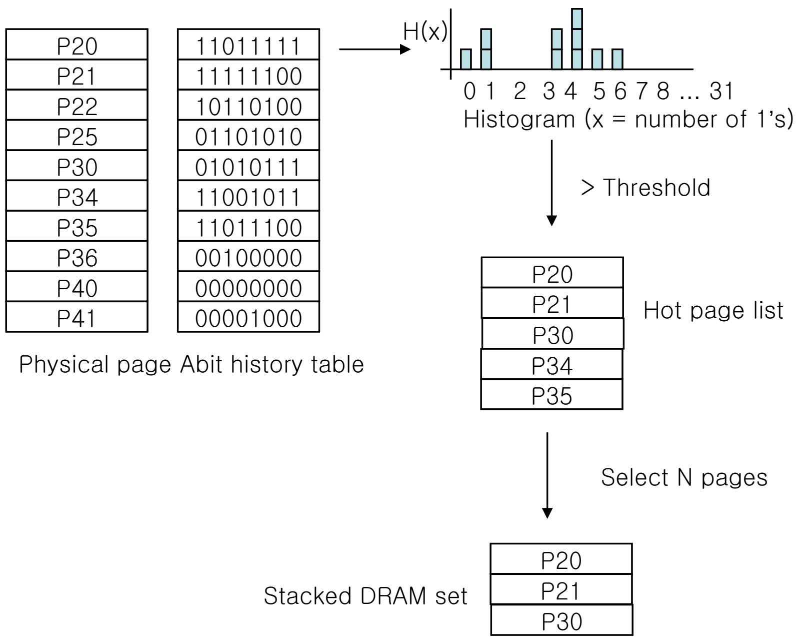


Figure 62: Tracking hot pages using access-bits

Since application processes within a guest VM run in virtual time, it is important to consider the process's virtual time rather than wall-clock time for an accurate A-bit history. Thus, page tables are scanned over 'virtual' time which avoids unnecessary page table scans, providing more accurate detection of hot pages. Each time the 100ms boundary is crossed in virtual time, the page table is scanned and A-bits are collected. For accurate accounting, events such as timer tick (TIMER), address space switch (NEW_CR3), and vCPU switch (SCHEDULE) are taken into account. This allows us to track each process' virtual time and accurately detect its hot pages. Our implementation collects and maintains A-bit history for each machine frames for any guest VMs, including the management domain.

Figure 63 depicts two VMs — VM0 and VM1 – where each VM has processes Proc1,2,3 and ProcA,B, respectively. Three events, TIMER, NEW_CR3 and SCHEDULE, occur along its execution timeline. They correspond to the 10ms timer tick, the CR3 switch (process switch), and the VCPU switch, respectively. At these points, the actual time spent in execution is calculated and accumulated for each process. In this fashion, each process' virtual time is tracked, thus enabling the accurate detection of hot pages (this is because virtual time is the actual time spent running on each core). This allows us to track each process' virtual time and accurately detect its hot pages. Our implementation collects and maintains A-bit history for each machine frames for any guest VMs, including the management domain.

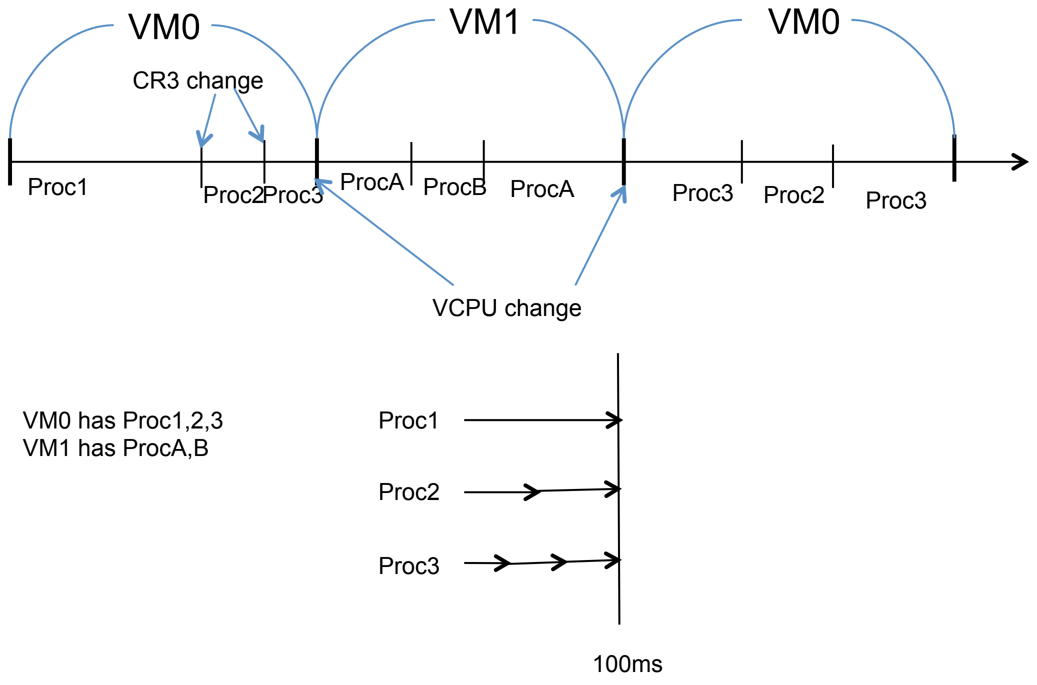


Figure 63: Tracking virtual time: TIMER event (10ms tick) is not shown.

Because the virtual time is utilized, A-bit histories from all active guest VMs are collected and maintained. Xen uses frame tables for physical memory management and this table is a large table that each entry corresponds to each physical page. This table is central to Xen's memory management and we extended this table to embed our A-bit history and other information as in Figure 64. This additional information such as A-bit history is used to save each frame's A-bit history. In addition, other fields such as next/prev pointers are used to form linked lists of pages for easy access.

Also, note that we have Rmap structure in this table, which saves reverse mapping information. It enables quick unmapping and mapping for any arbitrarily given pages. Each physical page, which mfn indexes, has one Rmap_list. Rmap_list is a list of Rmap_set and Rmap_set is a fixed sized array which contains pointers to page table (PT) and page table index (PTI). Thus, it provides reverse maps to the page tables so that it enables iterating all mappings to the given pages. By iterating these rmaps, virtual-to-physical mappings can be easily remapped. It would be too expensive operation without this Rmap structure. This remapping is used for page migration.

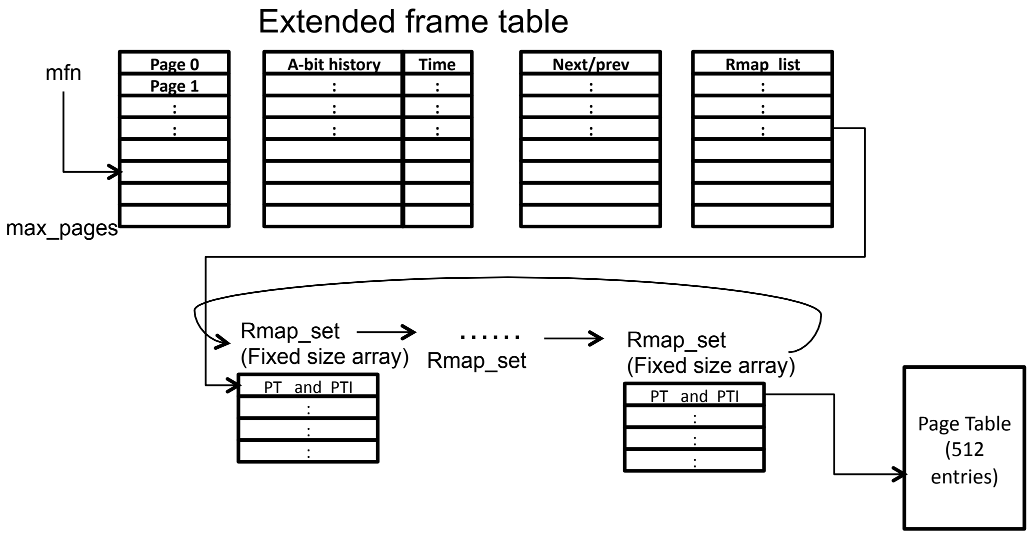


Figure 64: Extended frame table with a-bit history and reverse maps

To enable transparent page migrations to guest VMs, Xen is extended to mirror any guest's page table. Since the page table is installed in the hardware base register (CR3 in case of x86), any changes to CR3 is detected and page table is mirrored. This is quite similiar to shadow page table, but it is more flexible and also it was used to enable per-cache page tables in the earlier chapter. This allows us to change mappings without impacting guest OS.

4.2.1.1 Hot Page Management

Hot pages detected using A-bit history are actively managed by moving them in and out of fast/slow memories. There are four categories of pages and associated actions depending on their residencies and status, as shown in Table 6. Active hot pages should be migrated to or maintained in fast memory and inactive cold pages should be discarded. While cases 2 & 3 are relatively easy tasks, actions 1 and 4 are the primary determinants of the overhead of page migrations which are handled as described below.

Hot page lists can be quite long, which can result in substantial overheads to scan and migrate pages. In addition, inactive pages must be detected and evicted from the list. In total, there are four categories of pages, as shown in Table 6.

	Residency	**Status**	**Action**
1	Off-Chip	Active	Migrate to on-chip DRAM
2	Off-Chip	Inactive	Drop from the list
3	On-Chip	Active	Keep in on-chip DRAM
4	On-Chip	Inactive	Migrate to off-chip DRAM

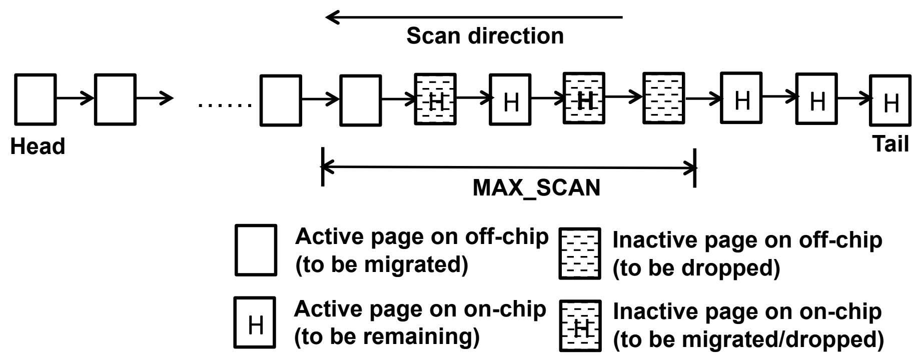
Table 6: Hot page management actions

Hot pages are managed to form a linked list (see Figure 65) which can be quite long if the workload has big working set and it can result in substantial overheads to scan and migrate pages. Thus, it is important to manage this list in efficient way. To efficiently manage this list and because page migration is expensive operation, only parts of the list are considered at one time where MAX_SCAN (currently 512) determines the number of pages that are scanned in a time window (every 100ms). This amortizes and limits overheads. Also this means that we do page-migrations somewhat lazily, not immediately and this seems to be good when working set is unstable. Further, the removal of inactive pages may incur page migrations to off-chip memory, causing significant overhead for other co-running applications. In addition, pages freed by the guest are likely to be used again by the next job since memory allocation in guest VMs often reuses previously freed pages. It is, therefore, beneficial to employ 'lazy' page migrations, delaying such page evictions from stacked DRAM, to reduce these overheads by migrating only MAX_MFNS pages from the hot page list. Also there is TIME_WINDOW macro that defines when pages in the list get inactive. TIME_WINDOW macro (currently 3000ms) defines when a page in the list become inactive. Thus, if a page in the list is not accessed for 3000ms, it is considered inactive and eventually discarded. So, if pages in the list dont get accessed for 3000ms, it would be considered inactive and eventually get removed from the list. While this removal is not an overhead, sometimes it may incur page migration to off-chip and this is expensive. Thus, the overall overhead largely depends on the number of page migrations. Therefore, lazy page-migrations help us to reduce some overhead here. Also, the host page list is scanned in reverse direction while newly arrived hot pages get inserted in head. Since the tail tends to populated with older pages, this optimizes further avoiding unnecessary page migrations.

In Table 6, while the categories (2) and (3) are a relatively easy task, categories (1) and (4) are the primary determinants of the overhead of page migration. It is these overheads that cause us to limit page migrations, as explained above and elaborated further next.

We currently move only some max number of inactive pages upon their detection with the timer-based approach described above. A more radical alternative we have considered is to evict pages from the list immediately upon an application's completion. However, this would incur significant page migration overhead likely visible to other currently running applications, plus there will be freed pages (by guests) that will again be used by the next job (because memory allocation in guest VMs often reuse previously freed pages). It is therefore, beneficial to delay such page evictions from stacked DRAM, where if pages are not reused, they will simply be migrated later, when detected as inactive.

A more radical alternative we have considered is to evict pages from the list immediately upon an application's completion. However, this would incur significant page migration overhead likely visible to other currently running applications. In addition, pages freed by the guest are likely to be used again by the next job since



Scan direction
Head
Tail
MAX_SCAN
Active page on off-chip (to be migrated)
Inactive page on off-chip (to be dropped)
Active page on on-chip (to be remaining)
Inactive page on on-chip (to be migrated/dropped)

Figure 65: Hot page management and associated actions

memory allocation in guest VMs often reuse previously freed pages. It is, therefore, beneficial to delay such page evictions from stacked DRAM, where if pages are not reused, they will simply be migrated later, when detected as inactive.

A final note concerning list management is that list scanning happens in a reverse direction, because newly added pages are added to the front of the list. So, the tail of the list typically has the oldest pages.

4.2.1.2 Transparent Page Migration

The region framework may migrate pages between different memories. Such migrations require remapping guest page tables which are hidden from the hypervisor. In order to do this in a guest-transparent way, guest's page tables are *mirrored* in the hypervisor as shown in Figure 66. For para-virtualized guests, page tables are write-protected and require hypervisor's involvement in updating page table entries through a hypercall. We simply intercept these calls and re-create a mirror version of the page tables and install them in the CR3 hardware register, forcing guest to use these mirrors. This allows us to freely change virtual-to-physical mappings, without any changes to the guest OS.

In addition, more optimizations are introduced to minimize negative cache impact. Rather than scanning page tables completely, separate metadata structures using linked list and pointers, shown in Figure 66, are used for easy iterations over the page table entries, optimizing page table scanning for access bits. Without this optimization, the whole 4KB of each page table would be scanned, thus trashing valuable 4KB-worth cache lines. This is especially true for interior (L2, L3, L4) page tables. Only existing mappings are managed in this list which effectively eliminates unnecessary scans. Furthermore, L1 page tables (leaf node) uses a bitmap to quickly detect present pages. This again eliminates unnecessary scans on the L1 page table and prevents valuable cache lines from being evicted.

Also, more optimizations are introduced to minimize negative cache impact. Rather than scanning full page tables, we use separate data structures using pointers for easy iterations over page table entries. Further, bitmap is adopted to easily find present pages in L1 page table. Without these optimizations, page scanning becomes too expensive, reading multiple 4KB page tables, and it would also trashes cache contents, which would be hidden overheads. We could minimize negative impacts on caches using such various optimizations.

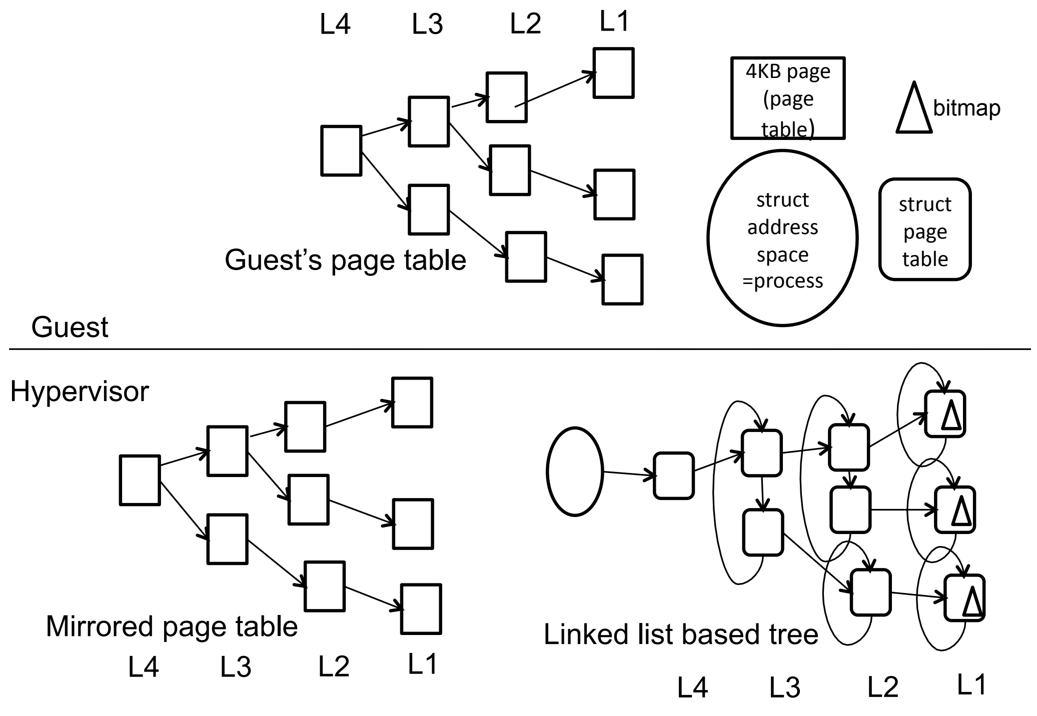


Figure 66: Page table mirroring and linked-list-based tree for quick scan

Further, a bitmap is adopted to easily find pages present in L1 page table. Without these optimizations, page scanning becomes too expensive, reading multiple 4KB page tables, and also evicting cache contents leading to further overheads. We minimize such negative impacts on caches using these various optimizations.

For guest transparency, the hypervisor mirrors each guest's page table and installs it in a hardware register. This allows us to freely change virtual-to-physical mappings, without any changes to the guest OS. In addition, small, separate, meta-data on these page tables are maintained, using a linked list and pointers, as shown in Figure 66. This optimizes the page table scan for access bits. Without this optimization, the whole 4KB of each page table would be scanned, thus trashing valuable 4KB-worth cache lines. This is especially true for interior (L2, L3, L4) page tables. Only existing mappings are managed in this list, so this effectively eliminates unnecessary scans. Furthermore, L1 page tables (leaf node) uses a bitmap to quickly detect present pages. This again eliminates unnecessary scans on the L1 page table and prevents valuable cache lines from being evicted.

4.2.1.3 Handling Shared Pages

Since any machine frame can be shared between multiple processes/guests, all the corresponding page table entries must be updated while migrating a page. To do this efficiently, we employ *reverse maps*, Rmap structure in Figure 64, which stores this reverse mapping information, i.e., from a page in physical memory to entries in various page tables, enabling easy remapping for any given page. We can iterate over this Rmap list to find all the mappings to a given page. Remapping pages would be prohibitively expensive without this Rmap structure. Thus, Rmap enables efficient page remapping. Each machine page (mfn) is associated with one Rmap_list, which is a list of Rmap_set. Rmap_set is a fixed size array that contains pointers to page table (PT) and page table index (PTI).

Xen hypervisor employs frame tables for easy memory management. This frame table is extended to have the A-bit history, Rmaps, and other additional information as shown in Figure 64. The last-update timestamps make it possible to detect inactive pages from the hot-page list while Next/Prev are pointers forming linked lists.

4.2.2 Memory Allocation Policy

Under the heterogeneous memory platform, the memory allocation is getting more challenging. In our experimental setup, we have a fast stacked-DRAM memory and slow off-chip DRAM. Our basic policy aims to place pages to utilize a limited capacity of fast stacked DRAM for a VM. Basically it moves the pages with highest hit rate to stacked DRAM. For hot read-only pages, we can maintain two copies in both memories – home copy and satellite copy – and this eliminates the need of migrating back the page to slow memory later. Such read-only pages can be simply discarded when it's evicted. However, dirty pages needs to be migrated back to the slow memory when it's evicted from the fast memory.

Regarding the allocation policy, similar to the physical memory allocation of OSes, we tries to aim to distribute stacked memory across VMs based on their activity. First, we can think about share-based allocation that uses some pre-defined shares. For example, such shares can be set by the administrator to divide the stacked DRAM capacity among VMs according to their policy. The memory can be distributed simply in proportion to each VM's pre-defined shares. For example, a VM with twice share would allocate twice stacked memory than other VM.

Second, the allocation may be based on the working set size to pursue overall performance improvement. This policy uses the working set size information for each VM control memory allocation. The working set size is used as the share value and the stacked DRAM capacity will be divided by a weighted sum formula.

4.3 Experimental Evaluation #1

4.3.1 Heterogeneous Memory Emulation

Many previous research works on stacked DRAM used some kind of architectural simulators. However, we chose to emulate heterogeneous memory on a real machine. On a multi-socket platform, we emulated the target heterogeneous memory to conduct the heterogeneous memory research on the actual systems with realistic server workloads.

Name	Normalized bandwidth	Normalized latency
M0	1	1
M1	0.42	2.5
M2	0.2	5.2

Table 7: Three PHP codes for dynamic page generation

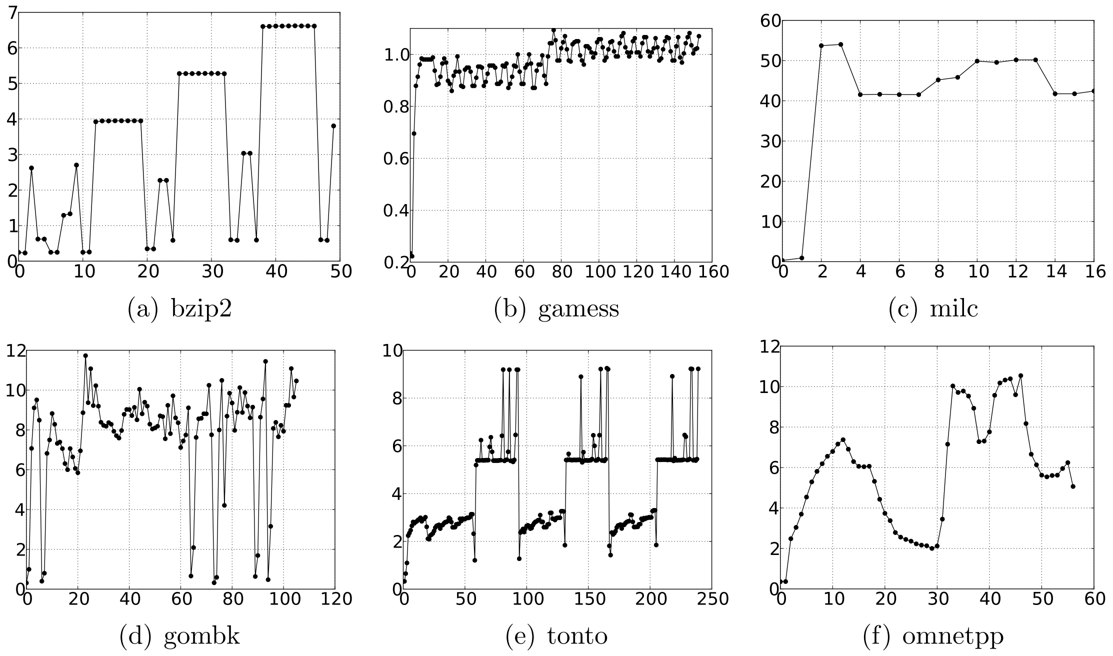


Figure 67: WSS curve for SPEC CPU2006 applications (x-axis = time (s), y-axis = WSS (MB)).

We emulate our heterogeneous memory system on a dual-socket 12 core Westmere x5650 server machine. It is equipped with 12GB DDR3 memory and the cores from

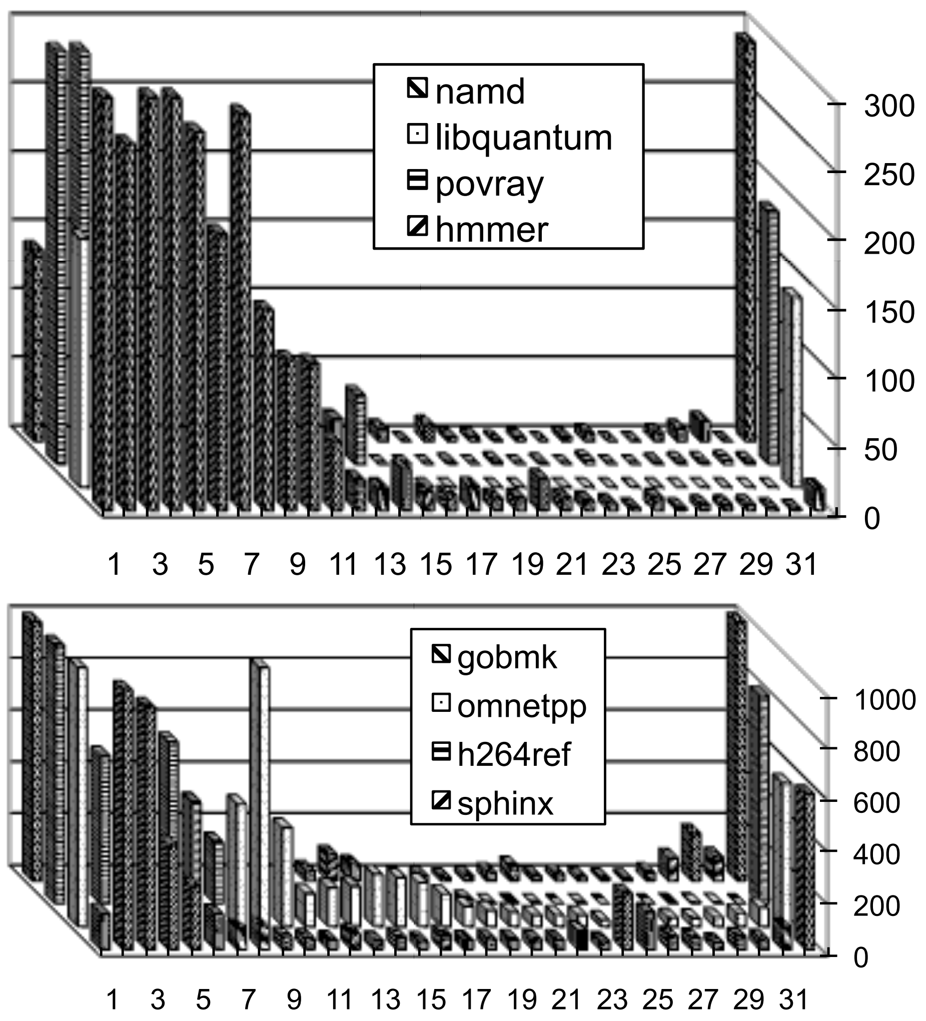


Figure 68: Histograms of pages and corresponding access counts

the second sockets are disabled, making only the cores from the first socket are used for running workloads. The workloads can access both memories at each socket but they experience various memory latency due to the NUMA configuration that provides an approximate 1.5x difference in memory latency between the two nodes. However, we further slow down the remote node using the memory controller throttling to emulate realistic heterogeneous configurations.

The memory throttling could be done by writing to the PCI registers (Integrated Memory Controller Channel Thermal Control). When applying different values of throttling, more various memory configurations could be emulated for other emerging memory technologies [100, 104]. See Table 7 for a comparison of normalized bandwidth and latency for three memory configurations for the memory-intensive stream benchmark [99]. M0 is no-throttling case and it serves as an base configuration for evaluation and M1 and M2 shows more throttled memory configurations. M1 represents small throttled case and M2 represents high throttled case. With higher levels of throttling, we can see lower bandwidth and higher latency with M1 (2.5x) and M2 (5x) configurations.

4.3.2 Workloads

We use a set of server-centric workloads of diversity to evaluate the impact of heterogeneity on server workloads. They are summarized in Table 8. They include CPU-intensive SPEC CPU2006 benchmarks, multi-threaded PARSEC benchmarks, and several MapReduce data processing benchmarks and with data analytics kernels. In case of the MapReduce benchmarks, it uses the shared-memory Phoenix implementation of MapReduce [102] and its input datasets are cached in memory.

4.3.3 Results

Figure 67 shows the working-set size (WSS) graphs as a function of time for several SPEC CPU2006 workloads. The WSS changes dynamically over time for almost all these applications, indicating the need for dynamic memory management. The working-set size represents only the amount of memory pages that are accessed by an application concurrently. However, it does not tell us how various pages are used by the current application in its allocated memory. So, even if two applications have the same WSS, each may have much diverse memory footprints. For example, one can use the same set of pages throughout its execution, while the other may keep changing its active memory region.

Figure 68 illustrates these workloads further. The figure shows the access count histogram for many SPEC CPU2006 workloads. The X-axis is the number of 1's in the access bit history and the Y-axis is the number of pages. Note that the maximum values of X axis and Y axis are set to 300 and 1000 for better visibility, respectively.

Looking at the figure, we can observe some clear distinction between hot pages and cold pages. Although some are in the gray area between the two, hot pages are located near the right end of the graphs and the cold pages are located near the left end of the graphs. Based on this observation, the hot page detection algorithm tries to capture the hot pages on the right side ($x > 22$) in Figure 68. We have decided experimentally the threshold value of 22 based on these observations. Some small changes to this threshold do not seem to make much difference, so the shown behavior was robust for different threshold values. These results verifies that the fact that at a given time only a fraction of the allocated memory is used actively by the application and those are critical for the performance. Thus, these pages should be placed in the fast memory for maximizing performance while the other cold pages may be placed in the slow memory.

Workloads Name	**Description**
Phoenix	Shared-memory MapReduce kernels
PARSEC	Multi-threaded application kernels
SPECCPU	Single-threaded CPU-intensive benchmarks

Table 8: Workload summary

As the memory is slowed-down, the performance dropped as expected for those workloads in Table 8. As expected, the performance dropped much severely in case of M2, compared to M1. The highest impact was for the mcf workloads to be 1431% (15x) and 537% (6x) for the two configurations, while many also showed an small impact. Thus, it is expected to see substantial performance gains by managing the hot memory pages for these workloads that experience severe impacts.

To evaluate the page migration mechanisms, the micro-benchmark *memlat* is used. Memlat allocates a large region of memory and access it randomly to measure memory

Time	**0 sec**	**5 sec**	**10 sec**	**15 sec**	**20 sec**	**25 sec**	**30 sec**
Memory latency (cycles)	900	792	620	498	367	367	367

Table 9: page migration evaluation using memlat

access latency. It verified that the memory latency goes down as the page migration mechanisms migrates hot pages from slow memory to fast memory as in Table 9. The latency initially was around 900 cycles and it gradually dropped to below 400 cycles in 20 seconds, while the latency remains high around 900 cycles when the migration is disabled. This verified that the page migration mechanism work and it helps lower the memory latency by page migration. This result shows the effectiveness of our software-controlled approach for managing heterogeneous memory.

4.4 Experimental Evaluation #2

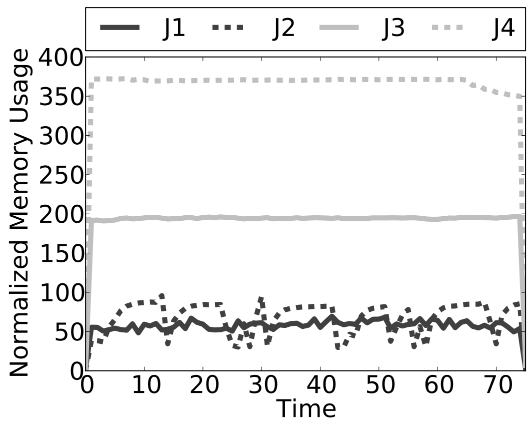
4.4.1 Experimental Setup

Our experimental platform consists of a dual-socket 12 core Intel Westmere server with 12GB DDR3 memory. In order to experiment with real platform and workloads, we emulate heterogeneity on this platform. Processor heterogeneity is emulated using CPU throttling by writing to CPU MSRs which allows changing the duty cycle of each core independently. Similarly, memory throttling is used to emulate heterogeneous memory (performed by writing to PCI registers of the memory controller) to slow down the memory controller. This allows us to experiment with memory configurations of various speeds such as M1 and M2 being approximately 2x and 5x slower than the original M0 configuration without any throttling.

Similarly, the incremental unit in the 3D-stacked memory is defined and it is termed E-state for convenience. The memory evaluation is done using a platform configuration with 512MB of 3D-stacked DRAM memory configuration and an E-state step of 64MB, resulting into 8 memory E-states. The fine-grained scaling mechanisms are compared against a base case configuration with static allocation of 1U CPU resources (E5 CPU state) and 256MB stacked memory (E4 memory state).

4.4.2 Workloads

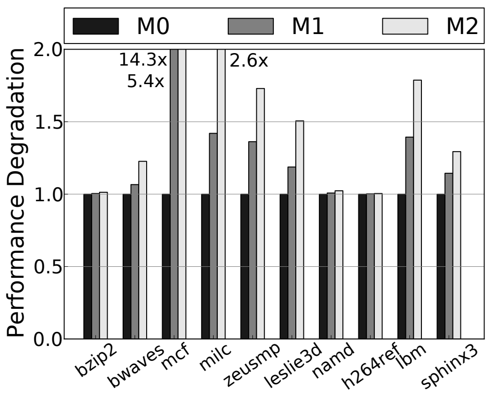
Evaluation is carried out using a web-server and an in-house memcached-like (memstore) application which service a stream of incoming requests from a client machine. The web server launches a CPU-intensive computation kernel, while memstore performs a memory lookup operation in response to each request. The memstore application allows us to load memory subsystem with its peak capacity, avoiding CPU and network bottlenecks associated with standard memcached implementation. In addition, several other benchmarks including SPEC CPU2006 and modern datacenter applications are also included in the analysis.



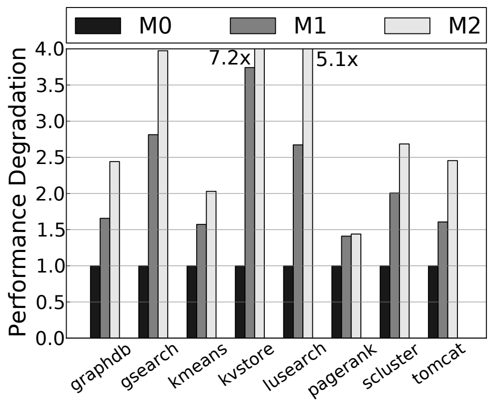
(a) Memory

Figure 69: Workload traces based on Google cluster data [59]

Evaluating the impact of memory heterogeneity, Figure 70 compare the performance of several SPEC CPU2006 applications (Figure 70(a)) and various modern cloud application benchmarks including graph database, graph search, key-value store, Lucene search engine, Tomcat server and kmeans, page-rank, streamcluster algorithms (Figure 70(b)) on different memory configurations. Specifically, it shows normalized performance at the base M0 configuration (without throttling) and M1 and M2 memory configurations (by applying different amount of memory controller



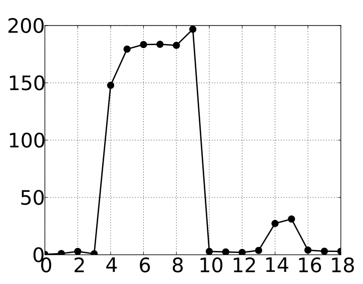
(a) SPECCPU



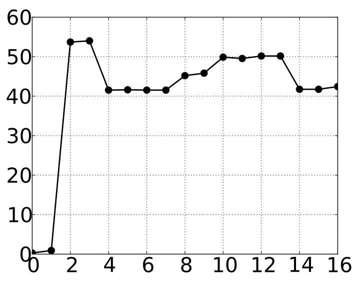
(b) DATACENTER

Figure 70: Impact of memory performance on several datacenter applications

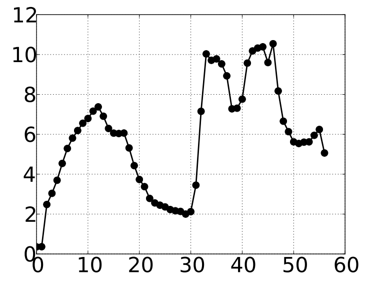
throttling). As evident from the figure, several applications observe severe performance degradation due to slow memory performance, including 14x (5x) and 7x (4x) performance loss for mcf and kvstore (key-value store) applications for the two memory configurations: M2 and M1. Other applications like bzip and page-rank exhibit less sensitivity. Overall, these results suggest that memory performance is critical to various applications which can benefit from the fine-grained management of heterogeneous memory resources.



(a) mcf



(b) milc

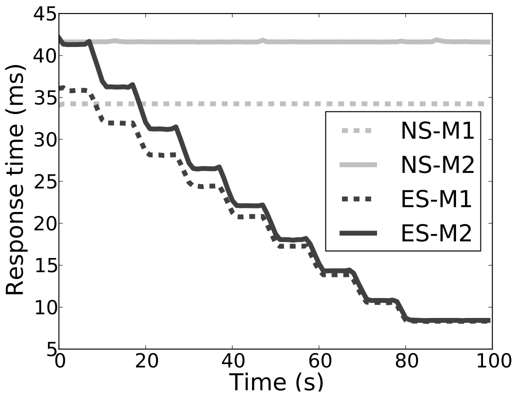


(c) omnetpp

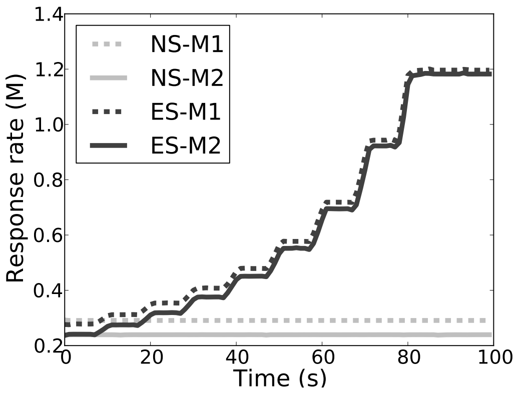
Figure 71: Working set size detection using access-bit history (x-axis = time (s), y-axis = WSS (MB)).

Showing the application of the A-bit history based mechanisms to obtain working-set sizes (WSS) of applications, Figure 71 plots WSS graphs as a function of time for several SPEC CPU2006 workloads. As seen in the figure, working-set sizevaries across applications from ~10MB for omnetpp to a much larger value of ~200MB for memory-intensive workloads like mcf. Further, WSS dynamically changes over time for these applications, thereby showing the need for dynamic memory management.

Further, Figure 72 shows performance impact of memory E-state scaling on the memstore application by gradually scaling the E-states from E7 to E0, i.e., increasing the size of fast memory allocation, where each state is maintained for five seconds before scaling to the next state. The non-scaling scenario (NS) shows a flat latency graph at 34ms and 42ms for the M1 and M2 configurations respectively. In comparison, when the E-states are scaled up from E7 (left) to E0 (right) in Figure 72(a), average latency for each memory operation decreases gradually to 8ms. The reduced access time with the fine-grained scaling also causes a 4.3x increase in the application throughput (from 0.28M to 1.2M) (Figure 72(b)). Also, the performance of the NS and ES configurations is comparable when no fast memory is used, signifying negligible overheads due to management operations such as page table scans, mirroring, and maintaining other data structures. These results demonstrate that resource scaling on heterogeneous memory systems can be applied to obtain the desired QoS for memory-sensitive applications.



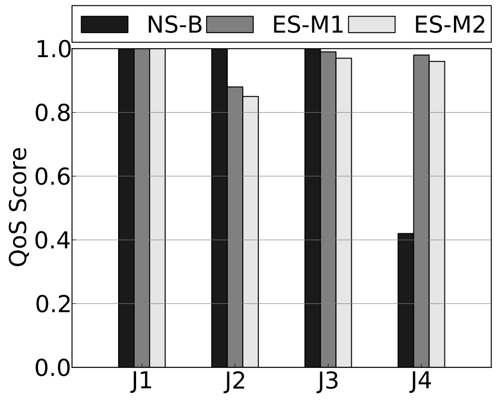
(a) Latency



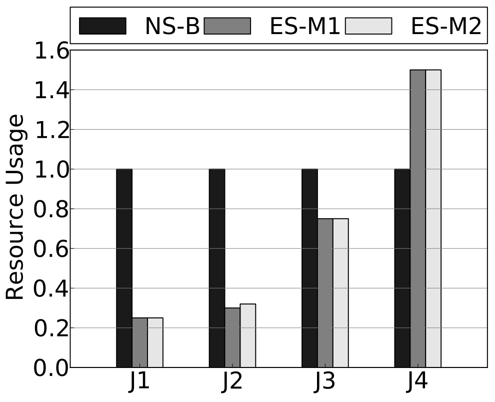
(b) Throughput

Figure 72: Impact of the fine-grained memory scaling on the performance of memstore application

Concerning fine-grained memory management, Figure 73 presents results for the



(a) QoS Score



(b) Resource Usage

Figure 73: Experimental results for memory E-state scaling

memstore application using the load traces shown in Figure 69 to vary the datastore size. The figure compares the QoS score, and resource usage for the base configuration (NS-B) with the QoS-driven policy (ES-Q) under M1 and M2 memory configurations. We also experimented with the resource-driven ES-R policy which showed only minor variation for the memory scaling experiments. The base case configuration consists of 256MB of stacked DRAM (state E4). As data in Figure 73(a) suggests, ES provides 2.3x better QoS score for job J4, while the performance is comparable for J1 and J3. J2 shows 15% performance loss with scaling due to its varying memory usage causing frequent scaling operations. Further, comparable behavior is noticed across the two memory configurations (M1 and M2). The resource usage results in Figure 73(b) illustrates that ES policies significantly reduce fast memory usage of jobs J1, J2, and J3 (75%, 70%, and 25% respectively). In comparison, J4 observes a 50% increase in its resource usage due to its large memory footprints. Overall, the fine-grained memory scaling using the region framework provides 30% lower stacked memory usage and thus cost, while maintaining performance.

In this manner, the region framework exploits platform heterogeneity and enables dynamic scaling of resources to meet desired application performance/cost trade-offs. As shown by the experimental data, it not only better services load peaks in comparison to homogeneous platforms (upto 2.3x) but also provides savings (average 21% for CPU and 30% for memory) scaling down resources during idle periods. Also, E-state driver can be customized to meet different user requirements, either meeting high QoS requirement using an aggressive policy or reducing resource usage while maintaining performance by using a conservative policy.

4.5 Related Work

For detecting the memory usage behavior of virtual machines, several approaches have been discussed including sampling, ghost buffer, etc. [103, 93, 98]. In comparison, the approach explored in this work uses page-table access bits to detect not only working set size of virtual machines, but also provide 'hotness' information of each page to guide page placement. Further, several architectural solutions have been proposed for page placement strategies for NVRAM-based hybrid memory systems, DRAM caches, and disaggregated memory [91, 101, 92, 95]. In comparison, our work focuses on software-controlled memory management for more efficient utilization of the stacked DRAM. Ballooning is a well-known technique for adjusting the memory allocation of virtual machines [42, 103]. this work generalizes such resource scaling in a unified way to various components. Given the rising contribution of memory power to overall system power, there is increasingly more emphasis on memory voltage scaling efforts [47]. Similarly, several studies have been made to understand the interaction between processor and memory voltage scaling [48]. This work's approach goes beyond voltage scaling and power management to support heterogeneous resources for efficient operation.

4.6 Conclusions And Future Work

This chapter presents systems software mechanisms for managing heterogeneous memory resources that consist of a combination of fast 3D die-stacked DRAM and off-chip DRAM. We believe that such stacked DRAM should be managed by software rather than by hardware (hardware managed cache) for flexible management. To this end, we propose and evaluate mechanisms for tracking the memory behavior of virtual machines and managing memory mappings, in a guest-transparent manner. We conduct experimental evaluations on an emulated heterogeneous memory platform. Preliminary results show the effects of memory heterogeneity on various workloads and our ability to track guest memory access patterns and improve performance by managing how stacked DRAM is used by applications.

Also this chapter presents the new memory management scheme using memory region for managing heterogeneous resources in the cloud platforms, providing fine-grained scaling capabilities for applications. To manage heterogeneity, it uses the memory region abstraction. The proposed abstractions are very useful to manage heterogeneity in memory subsystem. The region framework is implemented and extended in the Xen hypervisor. Evaluation is carried out using real-world traces on an emulated heterogeneous platform, showing that the region framework can provide VMs with the capabilities to quickly obtain resource for handling load spikes or minimize cost during low load periods.

There are several possible directions for future work. With the introduction of fine-grain resource scaling mechanisms on cloud platforms, a whole new area becomes open concerning the design of policies for requesting and allocating resources in the presence of multiple competing users. Market based allocation mechanisms based on game theory become relevant in this context.

Other form of heterogeneous memory could be small, fast scratchpad on chip. For example, single cloud computer (SCC) has a very small, fast scratchpad on chip. Such memory is more limited in its size, and faster. However, same pricinple of hot pages applies to the scratchpad memory. Hot data should be placed on the fast, small, memory such as scratchpad memory. Because it is more limited in its size, more sophiscated approach to choose hot pages may be needed. Also the big difference is that such scratchpad is typically private to the cores. This would create a new kind of region that is private to the cores. Meanwhile, NVM is also interesting case becasuse it has persistency. Perhaps a new kind of region such as non-volatile region can be created to capture the characteristics of being non-volatile.

However, there would be some limitations as well becasue the hot-page detection is inaccurate due to the large 4KB page size. It is possible that only one accessed cache line makes the whole page dragged into the fast memory wasting most of its space.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

In this thesis, the novel abstraction for memory-centric computing has been presented. It, combined with microscheduling, can be used to implement a cache-aware scheduler and guest-defined regions.

The final chapter of this dissertation summarizes what can be achieved with memory region, and identifies other open problems that can be research subject in the future.

5.1 The Memory Region

One of the major contribution of this dissertation is the introduction of the memory region abstraction and the region scheduler which employs the abstraction. Physical memory is partitioned into regions, and regions are mapped to caches. CPU cores can access only the allowed or mapped regions. This is enforced by the page tables used by the processor MMUs.

The basic idea of region scheduling is to move computation, not data, because data movement is expensive. Specifically, region scheduling causes the migration of computations to the caches in which regions are located. This migration is termed micro-scheduling, and the scheduler tries to find balance between high microscheduling overhead and performance benefits realized from data being local. Additional hardware support to optimize microscheduling overhead (e.g., to 100s cycles) would widen the usefulness of the approach. Finally, we note that as implemented now, region scheduling operates at page granularity, which makes region scheduling somewhat similar to a page-level approach to cache affinity.

5.2 Hardware Support and Codesign

The region framework is an implementation of a different execution model in which data is placed into memory units and computation moves to the data's location. This is in contrast to traditional approaches in which the hardware moves DRAM contents to cache, and cache lines to the processor's registers. We posit that such hardware driven data movements, shown useful in the era of single core computing, suffer from the high expense of data movements in the multi-core era. Cache coherency traffic further contributes to this problem.

To overcome the high cost of data movement, both in overhead and in power consumption, this thesis designs, implements, and experiments with the alternative memory-centric region-based execution model. Lacking hardware support for this model, however, results in its implementation solely in software, with consequent overheads due to frequent microscheduling and the need for hot page detection coupled with using the MMU for ensuring that processes always run where their memory regions are located. There are many possible codesign options that the overheads would be greatly reduced with.

(i) reduced microscheduling cost: the microscheduling is one of the major overheads and it is highly desirable that the hardware supports it directly. Some research papers assumes 100s cycles for context migration [34] while our pure software implementation showed tens of thousands cycles (47600 cycles or 17800 cycles depending on the machines). If such fast hardware microscheduling is supported, it would greatly broaden usefulness of the memory-centric scheduler.

(ii) efficiently detecting memory regions, at finer levels of granularity than the page-based methods used in regioning software: Current 4KB page is quite coarse to handle by the region framework and it is the source of inaccurate accounting and working set tracking. For example, the access bit tells us that at least one cache line in a 4KB page is accessed but it does not tell us how many cache lines are accessed. It may be any from one cache line to whole page that is accessed. This makes working set size tracking inaccurate. Furthermore, it confuses the regioning algorithm greatly if any unrelated object resides in a page because any access to the unrelated object is not differentiated from the regioning algorithm's perspective. Thus, allowing cache line granularity rather than 4KB page would greatly increase accuracy of the region framework algorithms.

(iii) make 'faults' on inappropriate memory references (i.e., accesses to memory in regions placed elsewhere) first class scheduling events. Currently per-cache page tables are used to enfore region-to-cache mappings and this can be implemented in hardware relatively easily. The pure software-based implementation adds a bit of overheads for page table manipulation operations and also it adds more TLB flushes. It also doubles space consumption as well.

(iv) detailed cache information: One of the considerations in the region scheduling is cache occupancy but it is hard to calculate it accurately. Software does not have information on which cache lines are evicted or not, and this makes it difficult to schedule tasks based on their cache footprint. Some research [123] proposes an useful hardware support for region scheduling and the cache footprints at cache line granularity would greatly help accurate task scheduling.

Region scheduling is implemented in the Xen hypervisor for generality, but this makes it harder to detect memory regions due to a lack of knowledge about memory page usage. The introduction of object to region mappings in Section 3.3 demonstrates the utility of gaining such additional semantic information. More specifically, as implemented now, regioning has to carefully detect and track which pages are stack, code, and data. This is done by at each page fault, analyzing the faulted address, the privilge mode, and the fault causes, to determine in what context of the OS the page is used. We were able to implement methods that determine page types, with high probability, but these methods strongly depend on the underlying architecture.

On the other hand, such information is readily available at the OS level, which suggest that (i) region scheduling may be more easily implemented in operating systems and/or (ii) there is a need for additional APIs between guest VMs and the hypervisor to assist the latter in determinations like these [128].

5.3 Conclusions and Future Work

This work started from the questions on the cache coherency protocols and the questions naturally led us to the reason why cache lines are moved around. The distance between the location of computation and that of operands is getting farther and it is also going non-uniform, moreover. The hardware approaches seems expensive in both performance and power, mainly because of lack of high-level information. The data is blindly moved around and, so, many side effects such as ping-pong effects are hard to avoid. We believe that software should be involved to properly solve this problem leveraging its high level information. Also we think that moving computation rather than data would make more sense in the future as the computation gets cheaper while the data movements gets more expensive.

Furthermore, new memory technologies such as PCM, NVM, 3D-stacked DRAM is posing new challenges. Hence, this dissertation proposes software approach to fulfill the new ideas and bridge between the hardware approaches and the software approaches for memory-centric computing. So, this dissertation introduces the memory region abstraction and its implementation on x86 to embrace many aspects of current and future memory subsystem. It was implemented in the hypervisor so that the unmodified guest OSes can run and benefits automatically from the hypervisor. The guest OS also may be written to use memory region services via hypercalls if it wants.

The region scheduling can be viewed as an alternative execution model based on the memory-centric paradigm. Traditional execution model is computation-centric and it moves around its data or operands between many other places such as memory, cache and registers. Although this was successful until now, it suffers from expensive data movements and scalability issues as the computing trends moves towards to multi/many-core environments. We believe that radically different approach is required to achieve scalable, low-power computing in the future and this new execution model or region scheduling could be the direction for the future computing platforms.

As the current hardware is not suitable for the new execution model, the new model had to be implemented in pure software level and it had to employ various tricky techniques to realize the ideas behind it. Microscheduling overhead was not so cheap that various optimization techniques are introduced to control the microscheduling overhead.

Also, using the memory region abstraction, functionalities could be mapped to each cache and cores could be specialized to the mapped functionality. The disaggregated OS services using region scheduling were demonstrated. This not only shows the usefulness of the memory region abstraction, but it also shows that the new execution model can be applied to the kernel space and potentially to the heterogeneous platoform.

The memory region is also showed to be useful to manage the newly emeged memory systems such as 3D stacked DRAM. The region framework is used to support the (emulated) 3D-stacked DRAM heterogeneous memory system. The basic idea is to identify hot pages across the whole system and place them onto 3D-stacked DRAM rather than slow off-chip DRAM memory.

Evaluation results show that the memory region can be very useful and the memory-centric scheduler is promising to achieve better utilization of caches. Also it could be applied to the kernel space, not only to the user space, showing that the cores could be specialized for specific functionality by using the region framework. Additional results show that the memory region can be used to deal with future heterogeneous memory systems such as 3D stacked DRAM.

To fulfill truely memory-centric computing, our new execution model should be codesigned with hardware approaches. Therefore, our future work is to codesign newly the memory-centric approach with hardware approach. Current implementation and design is for only current hardware, so the whole design and implementation should be reviewed if it is newly codesigned with hardware.

REFERENCES

[1] Larry Seiler, Doug Carmean, et al. "Larrabee: A Many-Core x86 Architecture for Visual Computing", SIGGRAPH 2008.

[2] Alexandra Fedorova, et al. "Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design." USENIX 2005 Annual Technical Conference.

[3] Vahid Kazempour, Ali Kamali and Alexandra Fedorova, "AASH: An Asymmetry-Aware Scheduler for Hypervisors," International Conference on Virtual Execution Environments.

[4] Ulrich Drepper. "What every programmer should know about memory," www.akkadia.org/drepper/cpumemory.pdf

[5] Sergey Zhuravlev, Sergey Blagodurov, Alexandra Fedorova. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". ASPLOS 2010.

[6] D. Tam, R. Azimi, L. Soares, M. Stumm. "Managing Shared L2 Caches on Multicore Systems in Software," Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), Jun 2007, pp. 27-33.

[7] L. Soares, D. Tam, M. Stumm, Int'l Symp. "Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-Only Pollute Buffer," on Microarchitecture (MICRO), Nov 2008.

[8] D. Tam, R. Azimi, M. Stumm. "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors," European Conf. in Computer Systems (EuroSys), Mar 2007.

[9] Mark S. Squillante, and Edward D. Lazowska. "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," IEEE Transactions on Parallel and Distributed Systems. Vol. 4. No. 2. February 1993.

[10] P. J. Denning. "Thrashing: Its Causes and Prevention." Proceedings AFIPS, 1968 Fall Joint Computer Conference, vol. 33, pp. 915-922.

[11] P. J. Denning. "Before Memory was Virtual." From the book In the Beginning: Recollections of Software Pioneers, edited by Robert Glass, IEEE Press, 1997.

[12] Huh, Kim, Shafi, Zhang, Burger, and Keckler. "A NUCA Substrate for Flexible CMP Cache Sharing," ICS 2005.

[13] Zhang, Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors", ISCA 2005.

[14] Evan Speight, Hazim Shafi, Lixin Zhang, and Ram Rajamony. "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors," ISCA 2005.

[15] Andreas Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," ISCA 2005.

[16] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in Micro, 2006.

[17] Lei Jin and S. Cho, "Better than the two: Exceeding private and shared caches via two-dimensional page coloring," in Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2007.

[18] D. Chandra, F. Guo, S. Kim, and Y. Solihin. "Predicting inter-thread cache contention on a chip multi-processor architecture," in HPCA, 2005.

[19] F. Guo and Y. Solihin. "An analytical model for cache replacement policy performance," in SIG METRICS, 2006.

[20] R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," in ICS, 2004.

[21] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin and C.Kozyrakis, "From chaos to QoS: Case studies in CMP resource management," in dasCMP, 2006.

[22] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in PACT, 2004.

[23] M. Qureshi and Y. Patt. "Utility-based cache partitioning: A lowoverhead, high-performance, runtime mechanism to partition shared caches," in Micro, 2006.

[24] Moinuddin K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs" International Conference on High Performance Computer Architecture (HPCA) 2009.

[25] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, Shalini Yajnik. "Supporting Soft Real-Time Tasks in the Xen Hypervisor," VEE 2010, Pittsburgh, PA, March 17-19, 2010.

[26] Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. "Characterizing application sensitivity to OS interference using kernel-level noise injection." Supercomputing08, November 2008.

[27] Paul Barham, et al. "Xen and the art of virtualization," In Proceedings of the nineteenth ACM symposium on Operating systems principles (2003), pp. 164-177.

[28] D. Rao and K. Schwan. "vnuma-mgr : Managing vm memory on numa platforms." In HiPC, Goa, India, 2010.

[29] Vishakha Gupta, Rob Knauerhase, Karsten Schwan, "Attaining System Performance Points: Revisiting the End-to-End Argument in System Design for Heterogeneous Many-core Systems" Sigops Operating Systems Review, January 2011.

[30] X10 Programming Language, http://x10.codehaus.org/

[31] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek, "Reinventing Scheduling for Multicore Systems," HotOS '09

[32] M. Lis, K. S. Shim, O. Khan, and S. Devadas. "Shared Memory via Execution Migration", ASPLOS Ideas and Perspectives Session, March 2011.

[33] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas, "System-level Optimizations for Memory Access in the Execution Migration Machine (EM2)," Second Workshop on Computer Architecture and Operating System co-design (CAOS), January 2011.

[34] M. H. Cho, K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Deadlock-Free Fine-Grained Thread Migration," Proceedings of the 5th Network-on-Chip Symposium (NOCS), May 2011.

[35] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, Fang Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," Proceedings of HPDC 2009.

[36] Gupta and Soffa. "Region Scheduling: An Approach for Detecting and Redistributing Parallelism" IEEE Transactions on Software Engineering Volume 16 Issue 4, April 1990.

[37] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (RaaS) cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[38] Amazon.com. Amazon Elastic Compute Cloud (EC2).

[39] Amazon.com. Amazon EC2 Auto Scaling.

[40] Amazon.com. Amazon EC2 Pricing.

[41] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14. ACM, 2009.

[42] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[43] L. A. Barroso. Brawny cores still beat wimpy cores, most of the time. *Micro, IEEE*, 30(4):20 –24, july-aug. 2010.

[44] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 273–286, New York, NY, USA, 2011. ACM.

[45] B. Cole. Samsung reveals big-little 8-core ARM for mobiles.

[46] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the eighteenth international conference on*

Architectural support for programming languages and operating systems, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[47] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: active low-power modes for main memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 225–238. ACM, 2011.

[48] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 143–154. IEEE, 2012.

[49] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10. IEEE, 2010.

[50] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12):48–57, Dec. 2009.

[51] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 263–270. IEEE Computer Society, 2012.

[52] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch. SOFTScale: stealing opportunistically for transient scaling. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 142–163, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[53] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In

Proceedings of the 8th USENIX conference on Networked systems design and implementation, NSDI'11. USENIX Association, 2011.

[54] Google. Google Compute Engine.

[55] P. Greenhalgh. Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7. White paper, ARM, Sept 2011.

[56] E. Grochowski, R. Ronen, J. Shen, and H. Wang. Best of both latency and throughput. In *Proceedings of the IEEE International Conference on Computer Design*, ICCD '04, pages 236–243, Washington, DC, USA, 2004. IEEE.

[57] M. Guevara, B. Lubin, and B. C. Lee. Navigating heterogeneous processors with market mechanisms. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 95–106, 2013.

[58] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. HeteroMates: Providing high dynamic power range on client devices using heterogeneous core groups. In *Green Computing Conference (IGCC), 2012 International*, pages 1 –10, june 2012.

[59] J. L. Hellerstein. Google cluster data. Google research blog, Jan. 2010.

[60] Y.-J. Hong, J. Xue, and M. Thottethodi. Dynamic server provisioning to minimize cost in an IaaS cloud. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 147–148, New York, NY, USA, 2011. ACM.

[61] Intel. Intel Xeon Phi Coprocessor - the Architecture.

[62] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the*

37th annual international symposium on Computer architecture, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM.

[63] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *High Performance Computer Architecture (HPCA), IEEE 16th International Symposium on*, pages 1–12, 2010.

[64] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 14–24. ACM, 2006.

[65] V. Kazempour, A. Kamali, and A. Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM.

[66] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multicore architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[67] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36. IEEE, 2003.

[68] W. Lang, J. M. Patel, and S. Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 47–55. ACM, 2010.

[69] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 451–462. ACM, 2012.

[70] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, 2010.

[71] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12. IEEE, 2012.

[72] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[73] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung. Challenges in heterogeneous die-stacked and off-chip memory systems. In *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*, Feb 2012.

[74] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 3:1–3:15, Berkeley, CA, USA, 2007. USENIX Association.

[75] R. Nathuji and K. Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 265–278. ACM, 2007.

[76] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 5–. IEEE Computer Society, 2007.

[77] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250. ACM, 2010.

[78] Nvidia. Variable SMP: A multi-core CPU architecture for low power and high performance. White paper, 2011.

[79] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[80] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '07, pages 289–302. ACM, 2007.

[81] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present and future. *Linux Symposium*, 2:223–238, 2006.

[82] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 99–110, New York, NY, USA, 2012. ACM.

[83] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[84] RightScale. RightScale Cloud Management.

[85] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *5th EuroSys*, pages 139–152, New York, NY, USA, 2010.

[86] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[87] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX conference on Operating systems design and implementation*, OSDI'02, Berkeley, CA, USA, 2002. USENIX Association.

[88] W. Wang, B. Liang, and B. Li. Revenue maximization with dynamic auctions in IaaS cloud markets. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–6, 2013.

[89] D. Wong and M. Annavaram. KnightShift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 119–130. IEEE Computer Society, 2012.

[90] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 21–30. ACM, 2009.

[91] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of SC*. IEEE, 2010.

[92] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. In *Proceedings of HPCA*, pages 1–12, 2010.

[93] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of ASPLOS*. ACM, 2006.

[94] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of ASPLOS*. ACM, 2012.

[95] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of HPCA*. IEEE, 2012.

[96] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings of ISCA*. IEEE, 2008.

[97] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung. Challenges in heterogeneous die-stacked and off-chip memory systems. In *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*, Feb 2012.

[98] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of ATC*. USENIX, 2007.

[99] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[100] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of MICRO*, pages 235–246, Washington, DC, USA, 2012.

[101] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of ICS*. ACM, 2011.

[102] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*. IEEE, 2007.

[103] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of OSDI*. USENIX, 2002.

[104] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *Proceedings of HPCA*, pages 1–12, 2010.

[105] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of VEE*. ACM, 2009.

[106] Livio Soares and Michael Stumm. FlexSC: Exceptionless system calls, Usenix ATC'11.

[107] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems, ASPLOS 2013

[108] Anshul Gandhi, Timothy Zhu, Mor Harchol-Balter and Michael Kozuch. SOFTScale: Stealing Opportunistically For Transient Scaling Middleware 2012

[109] Simon Malkowski, Yasuhiko Kanemasay, Hanwei Chen, Masao Yamamotoz, Qingyang Wang, Deepal Jayasinghe, Calton Pu, and Motoyuki Kawaba. Challenges and Opportunities in Consolidation at High Resource Utilization: Non-monotonic Response Time Variations in n-Tier Applications, In CLOUD 2012.

[110] Delimitrou, C., and Kozyrakis, C. Paragon:QoS-aware scheduling for heterogeneous datacenters. In Proceedings of the eighteenth international conference on Architectural support for programminglanguages and operating systems (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 77-88.

[111] VMWare vSphere. http://www.vmware.com/products/vsphere/

[112] XenServer. http://www.citrix.com/products/xenserver/overview.html

[113] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/

[114] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. ASPLOS2012

[115] MediaWiki. http://www.mediawiki.org/

[116] WordPress. http://wordpress.org/

[117] XpressEngine. http://www.xpressengine.com/

[118] Intel Smart Cache

[119] Changkyu Kim, Doug Burger, Stephen W. Keckler. "NUCA: A Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches", ASPLOS 2002

[120] Moinuddin K. Qureshi. "Adaptive Spill-Receive for Robust High-Performance Caching in CMPs", HPCA 2009.

[121] Min Lee, Vishal Gupta, Karsten Schwan. "Software-Controlled Transparent Management of Heterogeneous Memory Resources in Virtualized Systems." MSPC 2013 (ACM SIGPLAN Workshop on Memory Systems Performance and Correctness), Seattle, Washington, June 21, 2013.

[122] Joong-Yeon Cho, Hyun-Wook Jin, Min Lee and Karsten Schwan. "On the Core Affinity and File Upload Performance of Hadoop." DISCS 2013 (The 2013 International Workshop on Data-Intensive Scalable Computing Systems), Denver, CO, November 18, 2013.

[123] Mrinmoy Ghosh, Ripal Nathuji, Min Lee, Karsten Schwan, and Hsien-Hsin S. Lee. "Symbiotic Scheduling for Shared Caches in Multi-Core Systems Using Memory Footprint Signature." ICPP 2011 (IEEE International Conference on Parallel Processing), Taipei, Taiwan, September, 2011.

[124] Vishakha Gupta, Rob Knauerhase, Paul Brett, Karsten Schwan. "Kinship: efficient resource management for performance and functionally asymmetric platforms" CF '13 Proceedings of the ACM International Conference on Computing Frontiers Article No. 16

[125] Dulloor Subramanya Rao, Karsten Schwan. "vNUMA-mgr: Managing VM memory on NUMA platforms." HiPC 2010: 1-10

[126] Saraswat, Vijay; Bloom, Bard; Peshansky, Igor; Tardieu, Olivier; Grove, David, "X10 Language Specification Version 2.2"

[127] Jeff Bonwick "The slab allocator: an object-caching kernel memory allocator" USTC'94 Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1, Pages 6 - 6

[128] Vishakha Gupta, Rob C. Knauerhase, Karsten Schwan "Attaining system performance points: revisiting the end-to-end argument in system design for heterogeneous many-core systems." Operating Systems Review. 01/2011; 45:3-10.

[129] Benjamin Gamsa and Benjamin Gamsa "Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System" In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), 1999

[130] Patrick M. Widener, Matthew Wolf, Hasan Abbasi, Scott McManus, Mary Payne, Patrick G. Bridges, and Karsten Schwan. "Exploiting latent I/O asynchrony in petascale science applications." International Journal of High Performance Computing Applications, 25(2), May 2011

[131] Ada Gavrilovska, Karsten Schwan, Austen McDonald, Hailemelekot Seifu, and Ola Nordstrom. "Cooperative Application-level Processing on Hosts and their Attached Network Processors" Poster Session, 11th International Conference on Network Protocols (ICNP), Atlanta, Georgia, November 4-7, 2003

[132] Derek Murray, Grzegorz Milos and Steven Hand. "Improving Xen security through disaggregation" VEE 2008

VITA

Min Lee received his BS degree from Yonsei University in Korea and his MS degree from KAIST in computer science in 2004 and 2006, respectively. He is currently a Ph.D. candidate at the Georgia Institute of Technology. Among many other works, his region scheduling work, the memory-centric scheduling was published in ASPLOS 2012 and his research on low power in memory was published in IEEE Transactions on computers. He received the best paper award of 2006 Graduation at KAIST for his research on low power in memory and also received awards in the Samsung Humantech Thesis Competition and the 25th Student Thesis Competition held by the Korea Information Science Society. His other works include hypervisor-level real-time support for media server and hypervisor-assisted checkpointing, which were published in VEE 2010 and DNS 2011 respectively. His research interests center on systems area including operating systems, virtualization, and computer architecture. His current work focuses on heterogeneous memory scheduling in hypervisors.

Memory Region: A System Abstraction for Managing the Complex Memory Structures of Multicore Platforms

Min Lee

143 Pages

Directed by Professor Karsten Schwan

The performance of modern many-core systems depends on the effective use of their complex cache and memory structures, and this will likely become more pronounced with the impending arrival of on-chip 3D stacked and non-volatile off-chip byte-addressable memory. Yet to date, operating systems have not treated memory as a first class schedulable resource, embracing memory heterogeneity. This dissertation presents a new software abstraction, called memory region, which denotes the current set of physical memory pages actively used by workloads. Using this abstraction, memory resources can be scheduled for applications to fully exploit a platform's underlying cache and memory system, thereby gaining improved performance and predictability in execution, particularly for the consolidated workloads seen in virtualized and cloud computing infrastructures. The abstraction's implementation in the Xen hypervisor involves the run-time detection of memory regions, the scheduled mapping of these regions to caches to match performance goals, and maintaining region-to-cache mappings using per-cache page tables.

This dissertation makes the following specific contributions. First, its region scheduling method proposes that the location of memory blocks rather than CPU utilization is the principal determinant where workloads are run. It proposes a new scheduling method, the region scheduling that the location of memory blocks determines where the workloads are run. Second, treating memory blocks as first-class resources, new methods for efficient cache management are shown to improve application performance as well as the performance of certain operating system functions.

Third, explicit memory scheduling makes it possible to disaggregate operating systems, without the need to change OS sources and with only small markups of target guest OS functionality. With this method, OS functions can be mapped to specific desired platform components, such as file system confined to running on specific cores and using only certain memory resources designated for its use. This can improve performance for applications heavily dependent on certain OS functions, by dynamically providing those functions with the resources needed for their current use, and it can prevent performance-critical application functionality from being needlessly perturbed by OS functions used for other purposes or by other jobs. Fourth, extensions of region scheduling can also help applications deal with the heterogeneous memory resources present in future systems, including on-chip stacked DRAM and NUMA or even NVRAM memory modules. More generally, regions scheduling is shown to apply to memory structures with well-defined differences in memory access latencies.