Region Scheduling: Efficiently Using the Cache Architectures via Page-level Affinity

Min Lee, Karsten Schwan

Georgia Institute of Technology
Center for Experimental Research in Computer Systems
Atlanta, GA, USA
{minlee, karsten.schwan}@cc.gatech.edu

Abstract

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 paper 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. The paper 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.

Categories and Subject Descriptors D.4.1 [**Operating Systems**]: Process Management – Scheduling

General Terms Performance, Design, Experimentation

Keywords Virtualization, Xen, Cache, Memory, Region, Server Consolidation

1. Introduction

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 Intel's Dual-core Xeon chips) to 8 cores (e.g., in Intel's 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 mappings 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 paper 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 paper: (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 paper.

In order to make its performance advantages available to arbitrary applications and operating systems, region-based scheduling is implemented at hypervisor level, controlling VCPU to PCPU mappings and interacting with the hypervisor's page table structures (using the Xen open source hypervisor [27]). An alternative operating system-level implementation would apply region scheduling methods to the processes and their address spaces manipulated 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 VM's region mappings to caches; 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 implementation '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 Xen's 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 designing 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 scheduling offers system-level methods that improve and control application 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 paper, Section 2 describes the software architecture underlying the region scheduling approach, called the region framework. Section 3's analysis establishes the region tracking algorithm, and working set tracking is presented in Section 4. Section 5 presents performance evaluations. Section 6 details related work. In Section 7, conclusions summarize results and future work, including speculations on potential hardware support to reduce tracking costs.

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.

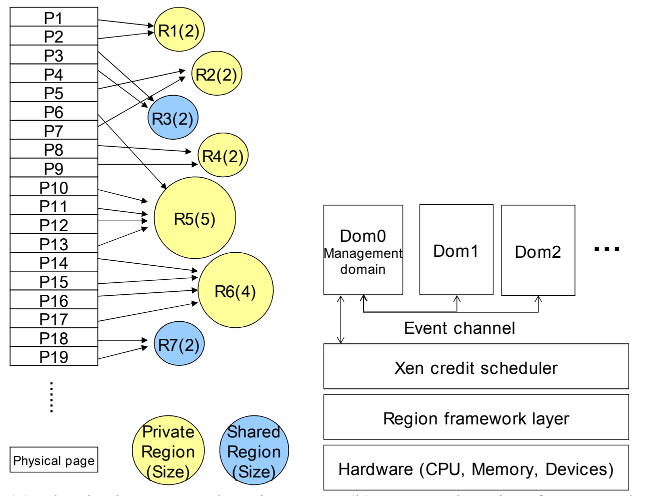
2.1 Software Framework and Methods

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.

Region-based scheduling explicitly places private regions into caches. Such mappings are maintained by having the scheduler restrict 'from where' the region's pages can be accessed, in accordance 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 ensure 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 4.1.

Figure 1(a) 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 1(b) illustrates this



(a) Physical pages and regions (b) Xen and region framework
(Size) is the number of pages that belong to (static) or the measured working set size at run time (dynamic).

Figure 1. Region framework overview.

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 entries) are created or destroyed and with changes in the behaviors of the executable entities using the address space. An address space's 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 4). C2(11) in Figure 2 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 2, 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 2 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 Region Scheduling – Implementation

Table 1 describes the data structure maintained for each region: (i)

Table 1. System-level representation of regions.

```
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
};
```

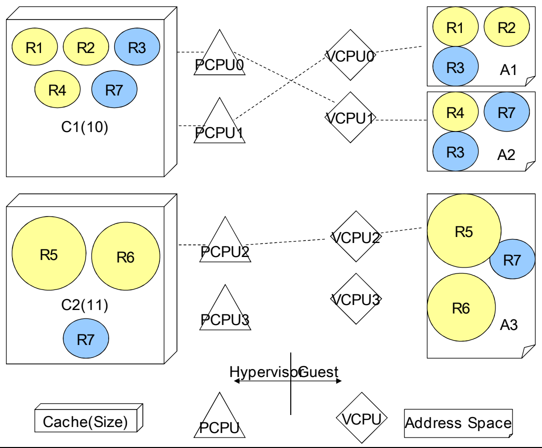


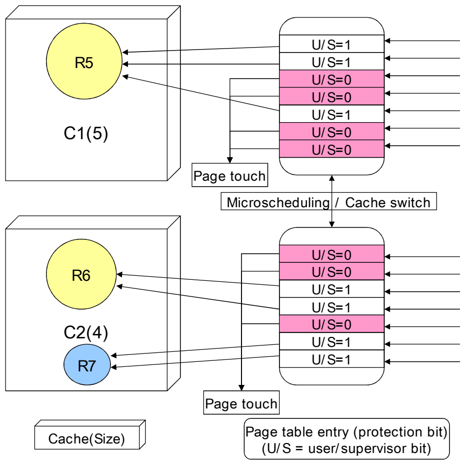
Figure 2. How regions interact with caches and address spaces.

if private, it belongs to a certain address space; (ii) it has a reference count, which is used to deallocate a region when it is no longer used; (iii) it is mapped to some number of caches (typically, to only one); (iv) it has a reverse map to the page table so that region to page mappings are easily changed; (v) there is additional information to calculate its working set size; and (vi) there are also additional counters for bookkeeping.

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. 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 3 depicts this. Note that without a reverse map, these actions would require an expensive complete page table scan.

2.2.1 Page touch and cache switch

As indicated in Section 2.1, 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


R5
C1(5)
U/S=1
U/S=0
Page touch
Microscheduling / Cache switch
R6
C2(4)
R7
Cache(Size)
Page table entry (protection bit) (U/S = user/supervisor bit)
Figure 3. Page touch and cache switch.

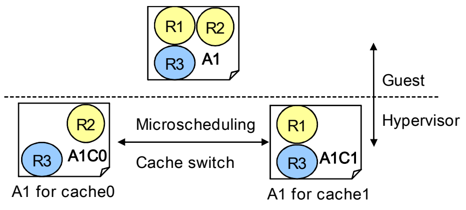


Figure 4. Per-cache page table.

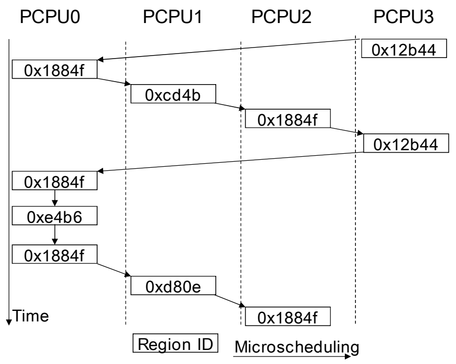


Figure 5. Microscheduling with 1VCPU.

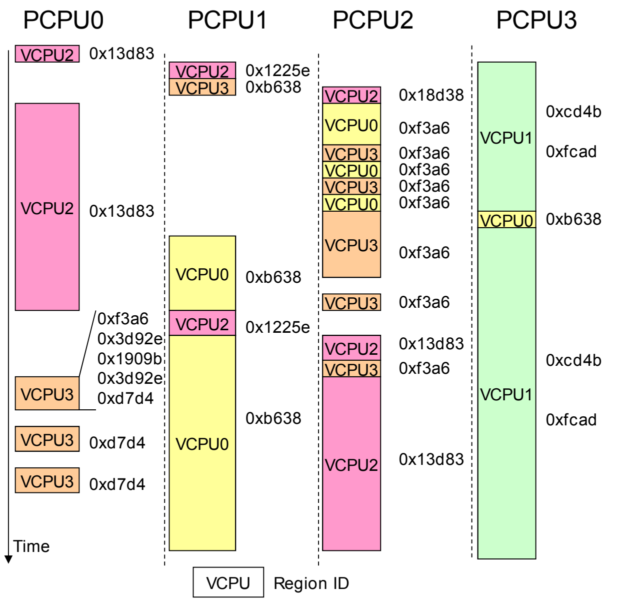


Figure 6. Microscheduling with 4 VCPUs.

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 other regions – 'closing' them. Figure 3 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 application's 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.

2.2.2 Micro-scheduling and cache switches

As evident from the description above, micro-scheduling involves cache switching. This could be expensive if it required the hypervisor to explicitly touch all of the address space's private regions and their page table entries. We eliminate this overhead by maintaining per-cache page tables. This is shown in Figure 4, where the hypervisor's page tables A1C0 and A1C1 jointly have the same contents as the guest's 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 concern VCPU/PCPU re-mappings. Figure 5 depicts a case in which one VCPU runs on four PCPUs, where 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 Xen's 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 6 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 7 for the associated region-to-cache mapping). We can see how the region framework balances cache loads from these figures. We discuss cache balancing in Section 4.

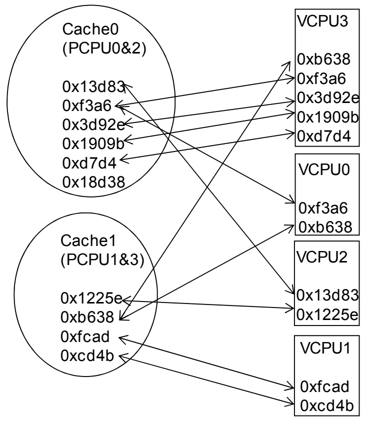


Figure 7. Mappings among regions, caches, and VCPUs.

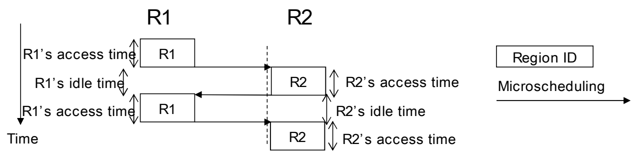


Figure 8. Two region memlat's access and idle times.

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 6, 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 competing for PCPU2, while PCPU1 is idle. This transient imbalance of VCPUs on two caches is due to restrictive region-to-cache mappings. 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 experienced 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 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.

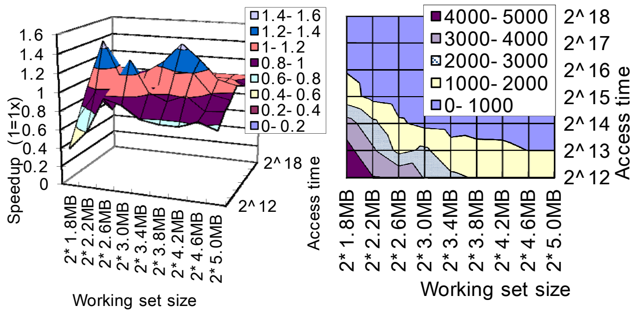
3. Regioning

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 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.

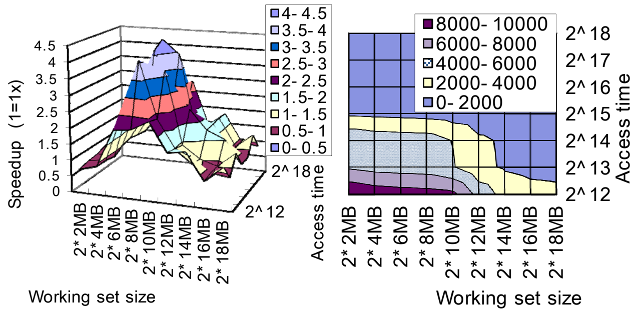
3.1 Cache Unification

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 8).

In the experiment, instead of confining the memlat's 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 5 for additional detail)



(a) Clovertown



(b) Westmere

Normalized performance (3D) and

(c) microscheduling rate per second (contour view)

Figure 9. Two-region memlat across caches.

Figure 9 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 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 performance improvements from using cache unification, up to 45% for Clovertown and over 300% for Westmere. This is despite significant numbers of micro-scheduling actions in Figure 9, 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 5.1). 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 increases as region size and access time decrease.

Second, on Clovertown, performance improvements are marginal when access times are high (>= 2^16) 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 (<=2^12), 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

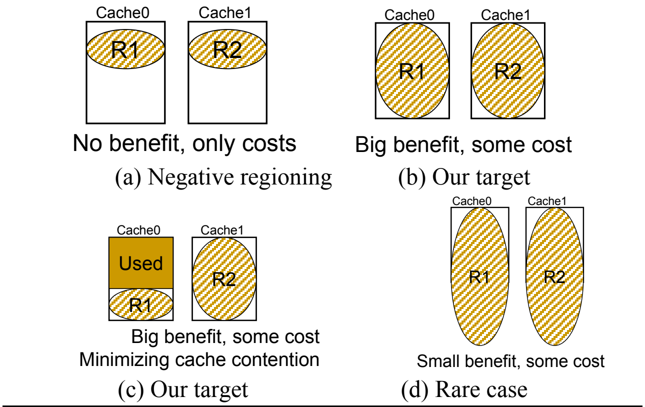


Figure 10. Cache working sets and cache sizes.

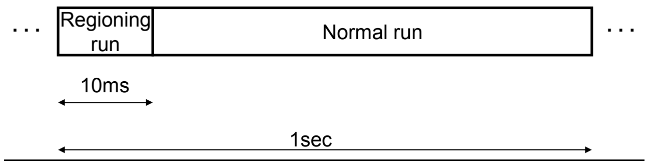


Figure 11. Regioning run (p = 1%).

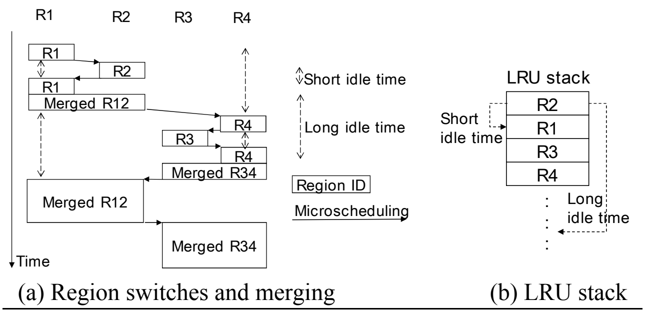


Figure 12. Regions and region merging via LRU stack.

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, 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 10. 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.

3.2 Bottom-up Regioning

We are now ready to explain how regioning is performed. Regions 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 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 11), 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 12 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 $\rightarrow$ 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.

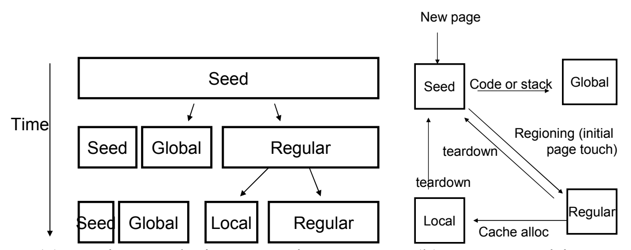
Idle times correspond to the LRU distance between regions. Therefore, a long idle time indicates a locality change, whereas a short idle time between two regions is a strong indicator for merging them, both of which are shown in Figure 12. Using a threshold 'q' to determine short idle times based on 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 regioning 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 2.2.2. As a result, the regioning phase can be entered by simply switching to separate regioning-phase page table that already has closed entries.

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 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 determined to be a 'stack' page. Both code and stack pages are classified into 'global' regions, and thus, do not further participate



(a) Region evolution over time (b) State transition

Figure 13. Types of regions.

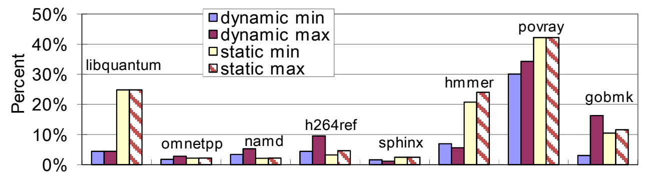


Figure 14. Portion of shared pages (observed min/max value).

Table 2. Regioning process.

(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.)

in the regioning process. The type transitions shown in Figure 13 apply to all other pages. In summary, the process in Table 2 is used for regioning.

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 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 multi-threaded 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.

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 glibc's cache. To address this issue, we declare all code pages (and similarly, the

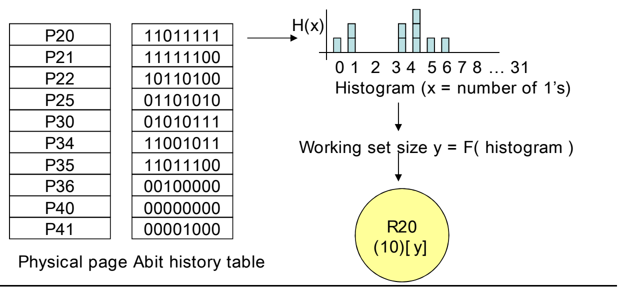


Figure 15. Access bit history and region's working set size.

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 14 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.

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 3, 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 page's 'access pattern' recorded as region histograms based on their pages' access histories.

Figure 15 shows region R20's details. From its 10 pages' access histories, it builds a histogram by counting the number of 1's, 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.

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

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 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 16 shows the evolution of cache working set sizes observed over time for select benchmark codes, which the cache balancer would use.

5. Experimental Evaluation

Region scheduling is evaluated on two generations of Intel platforms. 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 Data+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 two Intel six-core Xeon X5660@2.80GHz sockets with 12GB RAM. The caches are a 4-

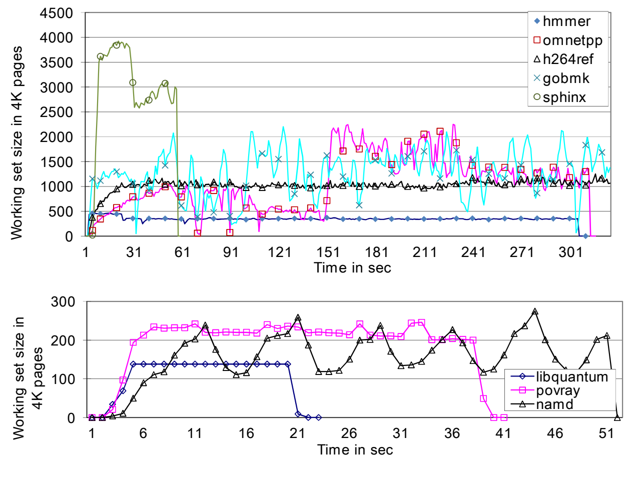


Figure 16. Tracking working set size.

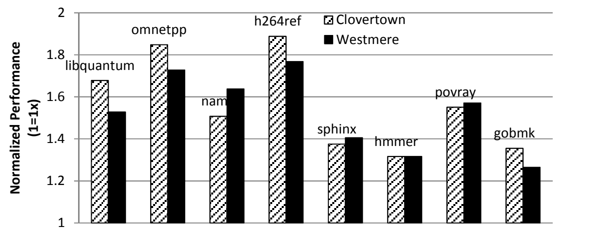


Figure 17. Speedup with cache balancing.

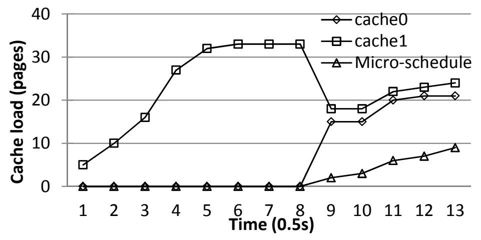


Figure 18. Cache-unification for two-region memlat.

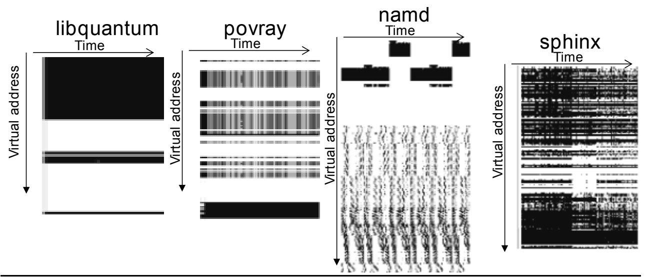


Figure 19. SPEC 2006 memory accesses.

way 32KB L1I, 8-way 32KB L1D, 8-way 256KB L2, and shared 16-way 12288KB L3.

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 machine's 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. 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 depending 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, overheads 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.

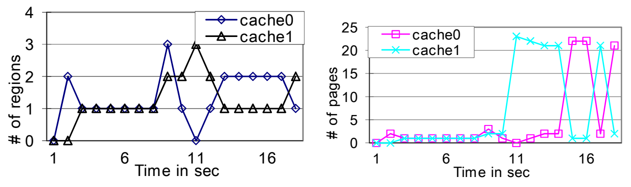
5.2 Microbenchmarks

To reduce cache contention, the cache balancer dynamically remaps 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.

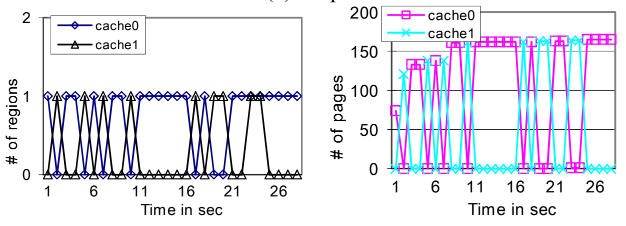
Figure 17 shows simple experiments on both machines, in which we run two 4MB memlats + 16KB memlat + a SPEC benchmark on the machine Clovertown, and two 12MB memlats + 512KB memlat + a SPEC benchmark on the machine Westmere. Depending on scheduling, the SPEC benchmark experiences different levels of cache contention. The cache balancer improves performance by correctly pairing processes onto caches and mitigating 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 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 18.

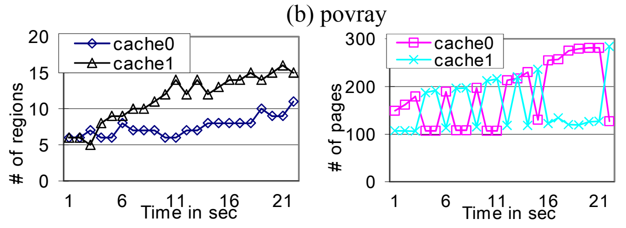
The first half of the figure shows unbalanced cache loads, and the second half shows balanced loads plus micro-scheduling. Similar results are obtained on machine Westmere and for brevity, are not shown here. We evaluate the performance implications of such actions in more detail below.



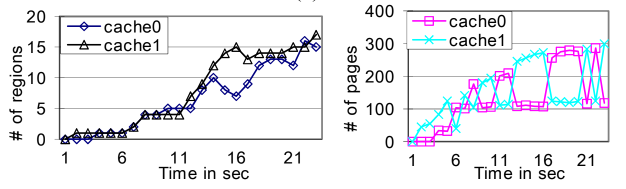
(a) libquantum



(b) povray



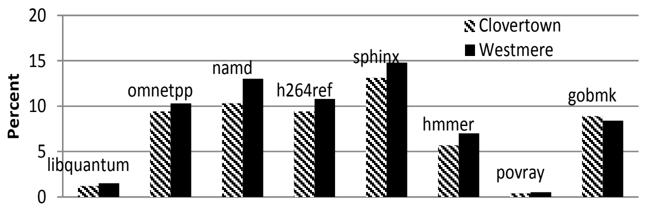
(c) namd



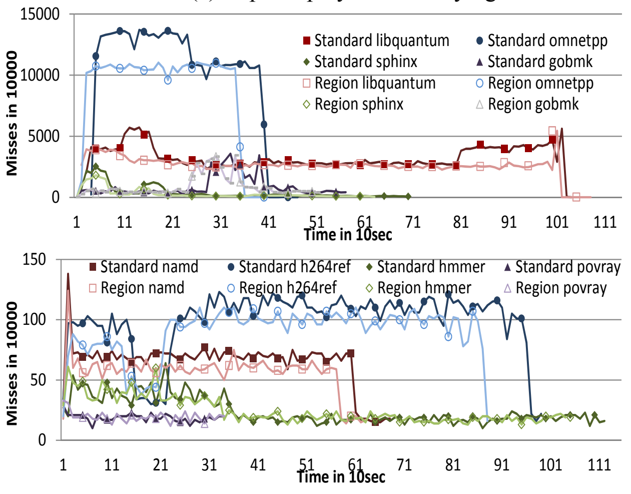
(d) sphinx

Regioning threshold = 100us on Clovertown

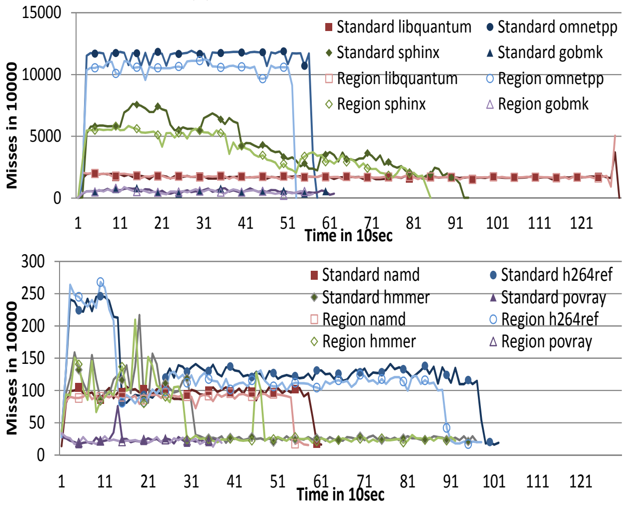
Figure 20. SPEC 2006 regioning.



(a) Speedup by cache unifying



(b) Cache misses for Westmere



(c) Cache misses for Clovertown

Figure 21. SPEC benchmark cache unifying.

5.3 SPEC Benchmarks

Figure 19 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 15). 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 20, 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 19, 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 20 (the graphs on the right) depict 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 21, 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 22, 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 per second! Further, Westmere shows better performance due to its cheaper micro-scheduling. The measured cache misses in Figure 21 demonstrate why this is the case. In many workloads, such as omnetpp, sphinx,

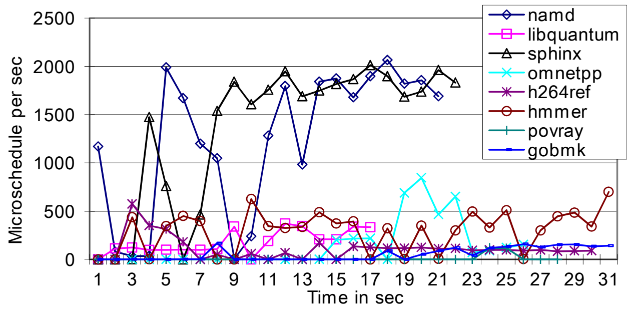
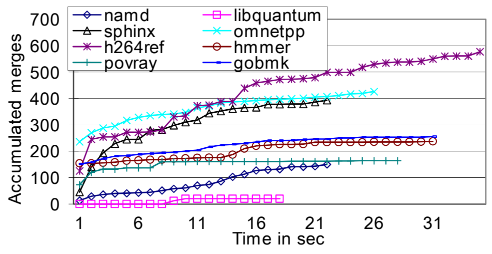
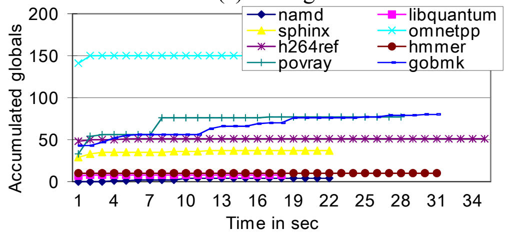


Figure 22. Micro-scheduling rates on Clovertown.



(a) Merges



(b) Globals (stack&code)

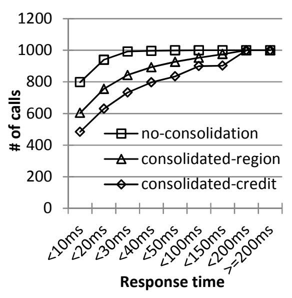
Figure 23. Regioning details (Clovertown).

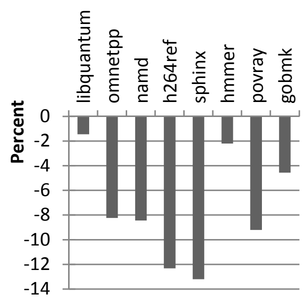
h264ref, the measured cache miss rates are lower with region-based scheduling.

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 relatively low cost context switching on modern architectures, there is an almost overwhelming importance of caching to program performance. 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 3.1, thereby demonstrating 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 scheduling.

Finally, the results in Figure 23 confirm that the bottom-up approach 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





(a) Response time **(b) SPEC cost**

Figure 24. Media server+SPEC consolidation on Clovertown.

stable to warrant their runtime detection and use, even without special hardware support for doing so.

5.4 Media Benchmark

Region scheduling can help improve performance, as shown in Section 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 communication 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 setup/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. 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.

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).

5.5 Cache Sharing

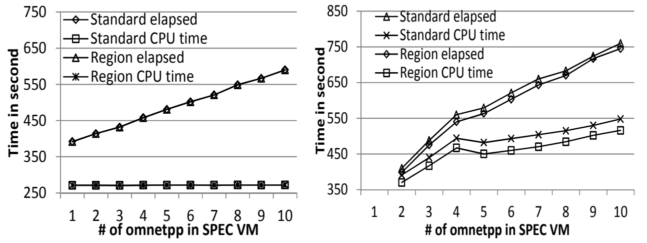
To demonstrate the utility of cache balancing, the VOIP experiment uses unfair policies that offer additional cache space to a preferred virtual machine. This is particularly important, of course, when there are multiple applications that share access to the platform's CPU and cache/memory resources.

Figure 24 shows the media server's 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 server's 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, respectively. 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.

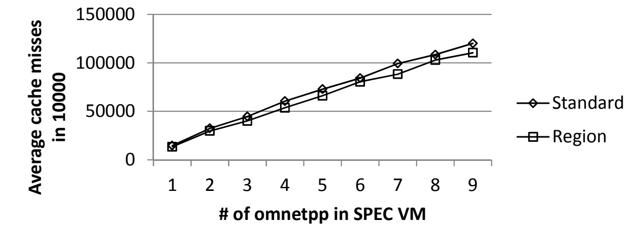
5.6 Reducing Noise for Parallel Codes

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 omnetpp workload (SPEC VM). Figure 25(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 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 omnetpp's performance suffers from sharing cache with PAR VM. Region scheduling improves the SPEC VM's performance by mostly isolating PAR VM onto one cache while SPEC VM runs on the other cache. Figure 25(b) shows their average cache misses. In general, with region scheduling, we observe 8.39% less cache misses in Figure 25(b).



(a) Performance for two virtual machines (Westmere)



(b) Cache misses (Westmere)

Figure 25. Parallel code VM consolidated with SPEC VM.

6. Related Work

The importance of efficient cache usage is well known [12, 13, 14, and 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 implemented 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 system's 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 paper.

7. Conclusions And Future Work

This paper 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 victim's 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 paper 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 being remedied in our current work.

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." Supercomputing'08, 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.