Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?

Lei Liu, Yong Li, Chen Ding, Hao Yang and Chengyong Wu

Abstract—On modern multicore machines, the memory management typically combines address interleaving in hardware and random allocation in the operating system (OS) to improve performance of both memory and cache. The conventional solutions, however, are increasingly strained as a wide variety of workloads run on complicated memory hierarchy and cause contention at multiple levels. We describe a new framework (named HVR) in OS memory management to support a flexible policy space for tackling diverse application needs, integrating vertical partitioning across layers, horizontal partitioning and random-interleaved allocation at a single layer. We exhaustively study the performance of these policies for over 2000 workloads and correlate performance with application characteristics. Based on this correlation we derive several practical rules of memory allocation that we integrate into the unified HVR framework to guide resource partitioning and sharing for dynamic and diverse workloads. We implement our approach in Linux kernel 2.6.32 as a restructured page indexing system plus a series of kernel modules. Experimental results show that our framework consistently outperforms the unmodified Linux kernel, with up to 21% performance gains, and outperforms prior solutions at individual levels of the memory hierarchy.

Index Terms— Multicore, Cache, DRAM, Bank, Locality Analysis, Memory Management, Operating System.

1 Introduction

MEMORY resource sharing is fundamental to the performance of multicore machines. To date, the most common mechanism for memory (DRAM banks) and cache sharing used in commodity machines is based on address interleaving: the physical address of a memory request determines where in the last level cache (LLC) set or DRAM bank the request is to be serviced. In the past many years, this approach has been general and effective to meet the needs of conventional workloads. However, as more applications are run in parallel, they often exhibit conflicting memory access patterns. Therefore, the "one-size-fits-all" approach becomes more risky, as it causes inter-program perturbation, resource thrashing, poor memory/cache utilization, and consequently, degraded performance.

Many prior studies rely on OS level partitioning-based memory management strategies to horizontally partition either DRAM banks [11,18,19,23,26] or LLC sets [15,17,22,28,35] to mitigate the interference among different programs. However, these "horizontal" partitioning approaches have the following drawbacks: **(1) Targetting a single level in the memory hierarchy:** Due to the horizontal nature, all these approaches attempt to partition only one level of the memory hierarchy. Thus, memory contention and conflicts were not addressed simultaneously at multiple levels in the memory hierarchy; **(2) Single policy based management:** These existing approaches are largely single policy based, and memory allocation optimizations are typically confined within the same level in the memory hierarchy, failing to provide flexible and customized memory allocation for individual applications' sharing and capacity needs; **(3) Application obliviousness:** Wisely using shared resource requires an understanding and differentiation of different applications' characteristics. Most prior studies show how to adapt a single strategy horizontally but not how to select the best strategy by considering both the memory hierarchy information and application characteristics.

To address all of the above challenges, we propose an OS based approach to partition and manage memory resource vertically across different levels of the memory hierarchy (e.g., DRAM and LLC). Enabling such *vertical* partitioning addresses contention at multiple memory levels and creates a larger solution policy space from which OS can choose to satisfy diverse resource requirements. In practice, there are still many cases where simple random interleaving allocation strategy performs better than any of the partitioning approaches. Therefore, in this work, we integrate **H**orizontal partitioning, **V**ertical partitioning, and a **R**andomized page interleaving policy (similar to H. Park et al.'s M^3 [27]) to create **HVR**, a low overhead, unified memory allocator that we implement in the

- *Lei Liu, Hao Yang and Chengyong Wu are with the State Key Laboratory of Computer Architecture (SKL), Institute of Computing Technology (ICT) Chinese Academy of Science (CAS). 0612J, No.6 Kexueyuan South Road Zhongguancun,Haidian District Beijing,China. E-mail: {liulei2010, yanghao2014, cwu}@ict.ac.cn.*
- *Yong Li is now with the vCenter Performance, VMware R&D 3401 Hillview Ave, Palo Alto, CA 94304. E-mail: yong1222@gmail.com.*
- *Chen Ding is with the Computer Science Department, University of Rochester, Rochester, New York. E-mail: cding@cs.rochest.edu.*

Linux kernel. HVR uses an online classification module to characterize memory/cache behavior and automatically selects among the candidate policies.

To determine appropriate memory management policies and to address conflicting allocation preferences, we conduct more than 10,000 experiments for over 2000 workloads on production machines with mainstream processor and memory configurations. Using data mining to analyze our results, we generate a set of practical partitioning and coalescing rules and organize them in a policy decision tree to enable HVR with the automatic policy selection, dynamic resource partitioning and coalescing. We summarize our contributions below:

(1) Vertical Partitioning (VP) (Section 3). We leverage the overlapped bits (O-bits) in the physical page address for indexing both DRAM banks and cache sets to partition memory hierarchy vertically, and achieve accumulated gains from multiple horizontal partitioning methods.

(2) SysMon (Section 4). We develop a kernel module tool named SysMon to dynamically capture application behaviors including memory footprint, active pages, page re-use time and cache utilization. The information is used to categorize applications online without offline profiling and without using hardware performance counters.

(3) Partitioning and Coalescing (Section 5.1 and 5.2). By mining extensive experimental results on varying policies for many workloads, we derive partitioning and coalescing rules to partition resources when needed, while allowing non-interfering programs to share resources.

(4) A Multi-Policy Framework (Section 5.3 and 5.4). We design the HVR framework that supports vertical partitioning, horizontal partitioning and random interleaving. We further design a variant VP approach (e.g., **Curve-VP**) that extracts more performance gains in common cases by combining partitioning and coalescing rules.

(5) Implementation in Linux kernel (Section 6). We introduce **x-Buddy**, a restructured physical page indexing system in Linux kernel 2.6.32 to support all the component policies and the partitioning/coalescing rules in HVR. The SysMon tool is implemented as kernel modules. Our implementation adds less than 3000 lines of source code into the Linux kernel source tree. HVR requires no hardware changes and performs well for both multi-threaded and multi-programmed workloads.

Our experiments on a real machine show that vertical partitioning outperforms prior techniques [17,18]. Based on the classification module (which is 100% accurate as verified by static profiling) and partitioning/coalescing rules (around 90% accuracy verified by experimental results), HVR system achieves consistent performance gains over the unmodified Linux kernel and outperforms prior utility-based software partitioning [17,18] by up to 21%. HVR is also demonstrated to perform well when handling dynamic workload changes in daily computing and production environments.

2 Backround

2.1 Buddy Memory Allocation System

Today's Linux operating system adopts a buddy system

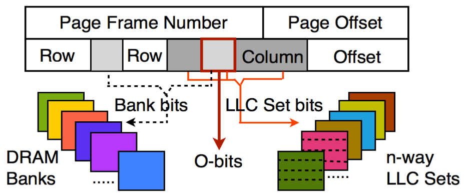


Figure 1. Address mapping for cache and bank partitioning.

to manage and allocate physical pages to various applications for low overhead and high efficiency [14,34]. The current buddy system maintains 11 free lists with *orders* ranging from 0 to 10. The free list with order *R* organizes pages in blocks, and each block has 2^R continuous pages. Upon a memory allocation request, the buddy system is responsible for identifying a free list with an appropriate order and selecting one block from the free list for allocation. One larger block with a higher order can be split into smaller blocks of lower orders when necessary. Buddy system aims to satisfy various memory requests from diverse applications as generally and efficiently as possible. To a considerable degree, the buddy system achieves its goal in the single-core era.

2.2 Page-Coloring Based Memory Management

Multicore architecture poses new system design and optimization challenges, particularly on memory allocations, since it allows all applications to share LLC (Last Level Cache) and DRAM banks, resulting in severe contention in many cases. Previous research efforts [13,26] show that contention can significantly degrade the overall system performance and many solutions have been proposed to mitigate the contention problem.

One of the most effective optimizations is page-coloring, which allows an OS kernel to leverage the underlying architecture information such as the physical address mapping of LLC and DRAM. With page-coloring, one can mitigate the contention problem [5,17,23,31,33,35,39] by modifying the kernel's buddy system while avoiding expensive hardware changes to memory controllers or cache hierarchies.

Page-coloring may be used for two purposes: cache partitioning and DRAM bank partitioning. As shown in Figure 1, cache partitioning can be achieved by using the bits in the OS physical page address that denote LLC set index (LLC color bits) as color bits. When allocating a page for an application, OS can assign a physical page with a specific color so that the application can only access the cache sets with the assigned color. Recent studies [11,18,19,23] utilize page-coloring to partition DRAM banks as there are also bits in the physical page address that denote DRAM bank address. The difference is that the bank color bits in physical page address might be distributed on some platforms [18,19].

3 Memory Allocation Policies

In this section we study the existing policies of memory allocation in the OS kernel and introduce our new methods to expand the policy space. Based on a performance analysis of these policies we identify several challenges and opportunities.

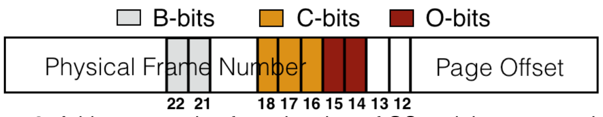


Figure 2. Address mapping from the view of OS and three categories of color bits on a typical multicore machine.

3.1 Memory Policies and Architecture Details

Traditional partitioning policies such as those mentioned in Section 2.2 are *horizontal* in that they partition either cache or main memory (DRAM banks) in one dimension. With the page-coloring technique, the horizontal partitioning can be implemented by selecting bank or cache indexing bits as colors when allocating a page. Our detailed architectural study reveals that the coloring bits can be classified into three categories: bank-only bits (B-bits), cache-only bits (C-bits) and overlapped bits (O-bits index both bank and cache in Figure 1). In particular, the O-bits enable a *vertical partitioning* (VP) that partitions both cache and memory banks, vertically through the memory hierarchy. By combining horizontal and vertical partitions, we can create many new policies.

Figure 2 illustrates the three types of color bits on a typical machine (Intel i7-860 with 8GB memory and 64 banks in 4 DIMMs. B-bits: 21~22; C-bits: 16~18; O-bits: 14~15). As the platform is equipped with two memory channels (denoted by XOR between the 6 and 16 bits [18, 19]), five bits (two B-bits, two O-bits and the 13 bit can constitute 32 colors) are sufficient to index 64 banks distributed across two channels (32 banks in each channel). This address mapping mechanism is called channel-level cache-line interleaving, which aims to maximize bandwidth utilization in common cases [19].

In our study, B-bits can be detected by prior approaches presented in [18,19,27], and C-bits can be inferred from [17] and Intel product manual. Notably, on our platforms, there are 7 bits (12~18 bit) in PFN that can be used to partition an 8MB last level cache. Among them, bits 12~13 also determine L2 cache index (particularly, bit 13 actually indexes L2 sets, LLC sets and DRAM banks), thus we do not use them in our partitioning techniques so as to avoid partitioning the L2 cache.

In Table 1, we list six representative policies using these coloring bits. Each policy partitions certain resources (i.e., cache, DRAM banks, or both of them) to a different extent and thus performs best in some of the scenarios. For example, A-VP uses the two O-bits to partition both LLC and DRAM banks into four colors (groups), thus it is suitable for applications with modest memory/cache demands as only one fourth of the LLC and DRAM banks can be used by each color.

3.2 Impact of Vertical Memory Management

Prior studies [9,11,15,17,18,22,23,26,39] demonstrate that horizontal partitioning on DRAM banks or LLC is effective in eliminating inter-program interference and improving performance. With vertical partitioning and, more generally, our partitioning policy space, one important question is ***whether the benefits from the horizontal memory and cache partitioning can be accumulated (i.e., should we go vertical in partitioning?).***

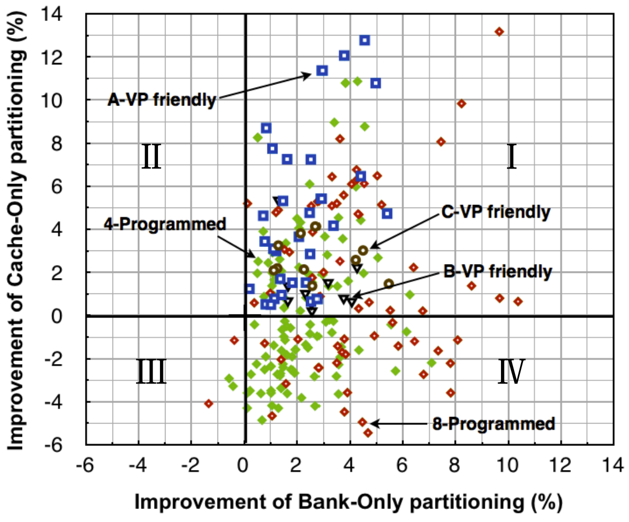


Figure 3. Performance improvement of Cache- and Bank-Only partitioning for 214 workloads. Green diamond dots: 4-programmed; Red diamond dots: 8-programmed; Blue squares: A-VP friendly; Black triangles: B-VP friendly; Brown circles: C-VP friendly. Note that the metric of overall system performance is Weighted Speedup, which is widely used to show the overall system performance.

Policy	Coloring Bits	Description	Target Cores
Cache-Only	C-bits {16~18}	LLC → 8 groups	4/8-core
Bank-Only (4)	B-bits {21~22}	Banks → 4 groups	4-core

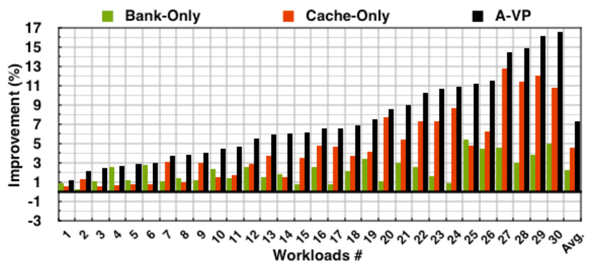
Bank-Only (8)	B-bits {21~22}
	O-bits {15}
		LLC → 2 groups
		Banks → 8 groups
			8-core

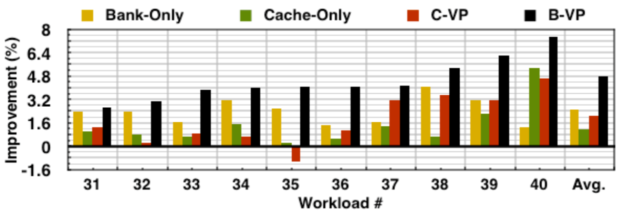
A-VP (4)	O-bits {14~15}	LLC → 4 groups Banks → 4 groups	4-core
B-VP (8)	B-bits {22} + O-bits {14~15}	LLC → 4 groups Banks → 8 groups	8-core
C-VP (8)	C-bits {16} + O-bits {14~15}	LLC → 8 groups Banks → 4 groups	8-core

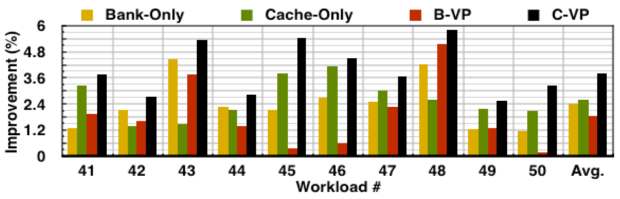
Table 1. Six representative partitioning policies.

To answer the above question, we investigate over 200 random workloads composed of programs from SPEC-CPU 2006 [1]. This experiment includes two steps. In the first step, for each workload we run multiple experiments to obtain the performance gains of that workload with horizontal partitioning. The performance improvements are compared against the unmodified Linux kernel as the baseline. All of the experimental results are reported in the four-quadrant Cartesian plane in Figure 3. The horizontal axis and vertical axis represent the weighted speedup [13] improvements achieved through the bank-only and cache-only partitioning, respectively. Workloads that contain 4 or 8 programs are denoted as 4/8-programmed workloads. From Figure 3 we can see about half of the tested workloads fall into Quadrant I. For these workloads, both cache-only policy and bank-only policies bring certain levels of performance improvements.

In the second step, we randomly select tens of workloads in the Quadrant I and measure their performance on A-, B- and C-VP, respectively. We find that these workloads (i.e., those highlighted by blue squares, black triangles and brown circles) achieve optimal performance with one of the VP policies (shown in Figure 4), indicating that their performance benefits accumulate to a certain degree due to the vertical partitioning on both cache and main memory banks. Shown in Figure 3 Quadrant I, dif-


(a) 4-programmed workloads (A-VP performs the best).


(b) 8-programmed workloads (B-VP performs the best).


(c) 8-programmed workloads (C-VP performs the best).
Figure 4. Performance improvement of A/B/C-VP.

ferent symbols denote workloads with different properties. For example, blue squares represent A-VP friendly workloads, which achieve the best performance with A-VP policy, as depicted in Figure 4 (a).

Quadrant IV contains workloads for which bank-only partitioning is beneficial while cache-only partitioning is detrimental. We study workloads in this quadrant and find that the performance benefits achieved by bank partitioning are largely offset by the side effect of cache partitioning (VP is not useful). Thus, for workloads in this quadrant, it is desirable to disable cache partitioning and enable bank partitioning. There are few workloads in Quadrant II and Quadrant III, indicating that bank-only partitioning is rarely harmful, and does not bring negative impact under most of the cases.

From Figure 4 we can draw the conclusion that for workloads that benefit from both cache-only and bank-only partitioning (50 workloads in Quadrant I of Figure 3), VP can accumulate the performance gains. For these workloads, using one of the A-/B-/C-VP policies can achieve optimal performance. For instance, A-VP can achieve up to 16.7% improvement over the baseline system (unmodified Linux kernel), while 5.9% and 11.7% over the cache-only and bank-only partitioning, respectively. We also run these multi-programmed workloads on random, interleaved page allocation (in section 3.3), and they perform better with the partitioning policies.

3.3 Random-Interleaved Allocation

Although the above analysis demonstrates that various partitioning policies achieve different levels of performance gains, there are cases where none of the partitioning-based memory allocation is preferred. One important scenario is that the running workload exhibits heavy data

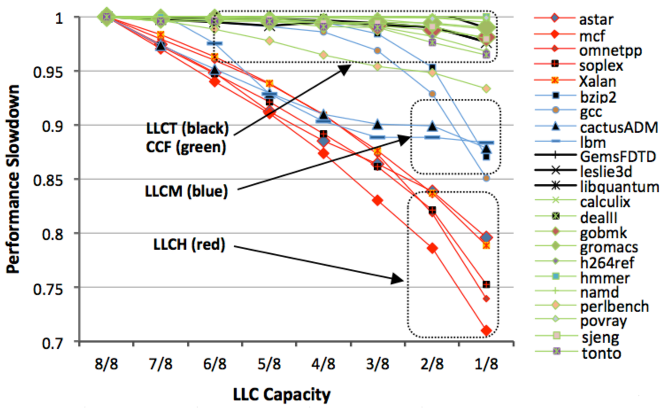


Figure 5. Normalized slowdown with different LLC capacity.

sharing, which defeats the purpose of any resource partitioning mechanism [18]. For example, multi-threaded workloads typically share considerable amount of data and thus multiple threads access the same memory or cache bank regardless whether the memory/cache is partitioned or not.

To optimize multi-threaded workloads, the recently proposed M^3 [27] enforces a random-interleaved page allocation in the OS kernel to avoid hot spots and row buffer conflicts on heavily shared banks. We conducted experiments and verified that the random memory allocator outperforms partitioning-based approaches for multi-threaded workloads. Therefore, to handle multi-threaded workloads, we integrate a randomized page-interleaving scheme to achieve similar effects of M^3 in our framework (see Section 6.2).

An obvious conclusion can be drawn from the above presented quantitative study is that the effectiveness of a memory allocation policy depends on specific application characteristics, in particular the cache requirements. In practice, a workload could contain several simultaneously running applications with an arbitrary combination of diverse characteristics, making it difficult to determine the appropriate policy of memory allocation.

4 Application Characterization via SysMon

Determining an advantageous memory allocation policy requires an accurate prediction of a running workload's memory/cache characteristic and its reaction to each allocation policy. To characterize application memory behavior, we develop ***SysMon***, an efficient online tool integrated in the Linux kernel to monitor system-level application activities such as page access frequency, memory footprint (all used pages), active pages, page re-use time, etc. Collectively, these metrics from SysMon can be used to classify applications into different categories and select appropriate memory allocation policies.

4.1 Application Categories

Based on our experiments we find that most multi-programmed workloads are not negatively affected by a modest bank-partitioning (<= 8 groups) scheme. By contrast, the performance of cache partitioning exhibits great variations due to different cache utilization behaviors of the running workloads (in Figure 3).

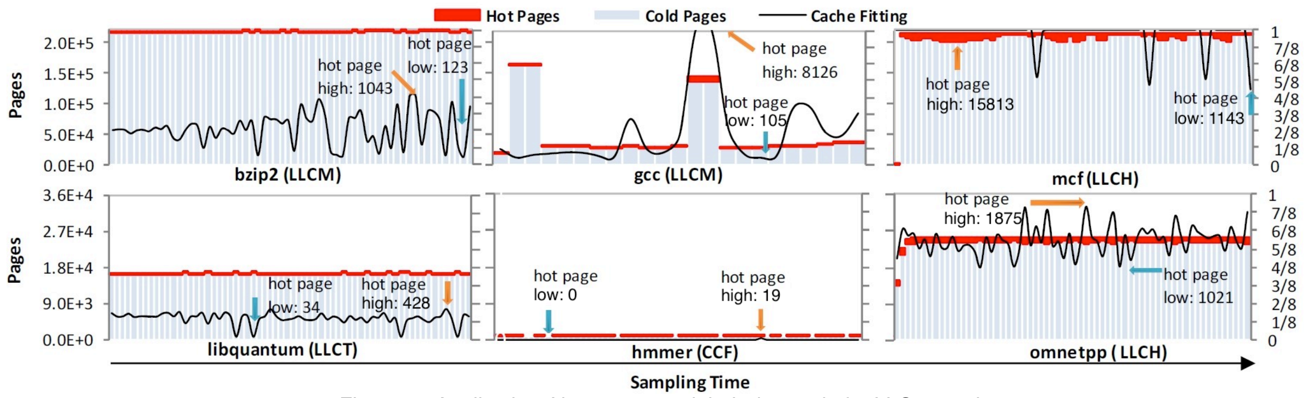


Figure 6. Applications' hot pages and their demands for LLC capacity.

In order to verify the potential impact of cache utilization characteristics on cache partitioning policies, we collect performance slowdowns of various applications as the cache quota is reduced from 8/8 (entire cache is used) to 1/8. Each application is executed eight times and each time a different amount of LLC is assigned by the page-coloring based cache partitioning. Based on the results we classify applications' caching behaviors into four categories: Core Cache Fitting (CCF), LLC High (LLCH), LLC Middle (LLCM) and LLC Thrashing (LLCT). Figure 5 reports the classification of various benchmarks in the SPECCPU 2006 benchmark suite [1]. CCF applications (denoted as green curves), such as *hmmer* and *namd*, do not degrade significantly when using fewer LLC resources since their working set sizes are small enough to fit into the L1 and L2 per-core private caches. LLCT applications (black curves), such as *libquantum*, are also insensitive to cache quotas, but due to cache thrashing behavior rather than small working set sizes. LLCH applications (red curves) such as *mcf* suffer the worst performance degradations from reduced cache quotas due to their resource hungry characteristics. Compared to LLCH, LLCM (blue curves) applications use fewer cache resource, thus the slowdowns are not as much as in LLCH applications. For example, *gcc* and *bzip2* are LLCM as they suffer no significant degradation when cache decreases from 8/8 to 4/8. However, a sharp performance drop is observed when the cache quota drops below 3/8.

4.2 Dynamic Application Classification

The static profiling approach used to generate Figure 5 requires running each application multiple times and does not capture dynamically changing behavior. To predict cache requirement and classify applications on the fly, we explore the relation between application page access behavior and cache utilization. ***The key insight*** is that number of *hot pages* (active pages used in a particular time interval) can reflect an application's LLC demand in many cases due to the DRAM row-buffer locality [29]. SysMon can identify hot pages by examining the access bit [39,40] in the page table entry (PTE). Figure 6 shows the correlation between the number of hot pages and cache demands for several benchmarks. Taking *hmmer* as an example, the number of hot pages, denoted in red color, is extremely low (at most 19 hot pages over the entire sampling period). This indicates that a maximum amount

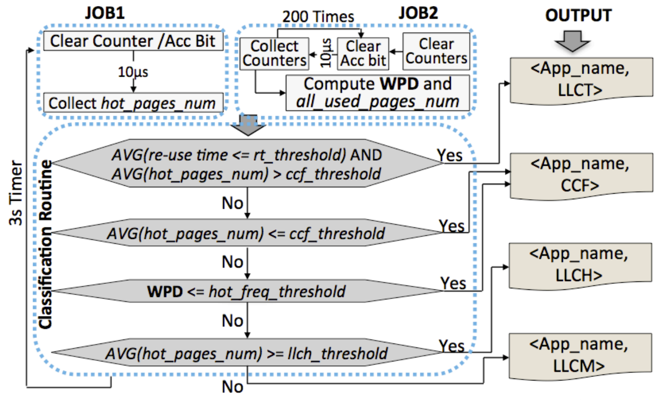


Figure 7. Online application classification algorithm in SysMon. (AVG(x) is a function to compute the average of x in three consecutive intervals).

of 19×4KB (4KB per page) cache resource is needed during the sampling period. Similar principle can be applied to other benchmarks in SPECCPU 2006.

To this end, a simple estimation of LLC utilization can be achieved by dividing the number of hot pages (NHP) by the number of pages the LLC can accommodate (NPC). This metric (NHP/NPC) represents the percentage of LLC occupied by hot pages and is shown as the cache fitting curve in Figure 6 (vertical axis on the right side of each sub-figure). In the case of *hmmer*, less than 1/8 of the LLC is required, indicating a CCF classification. A sharp contrast is *mcf*, which is classified as LLCH since it has large amount of hot pages (1143 to 15813) that cannot be accommodated in most modern computers' LLC (e.g., 8M). For an LLCM application (e.g., *bzip2*), the number of hot pages falls between that of the LLCH and CCF application and the required LLC quota typically varies between 1/4 and 1/2.

For an application that touches a large number of pages but exhibits poor reference locality (e.g., *sjeng* touches many hot pages but only a small amount of them are heavily accessed and thus would benefit from being cached), using only the number of hot pages will mislead the above simple method to a wrong classification of LLCM or LLCH. To address this issue, we define *weighted page distribution* (WPD), a metric used to reflect page reference locality and can be obtained by per-page access counters in SysMon. Based on the above analyses, we devise an online application classification algorithm detailed next.

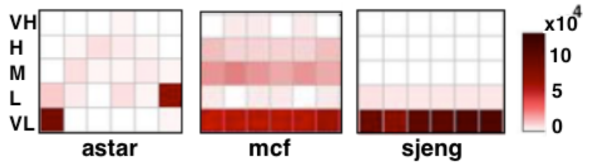


Figure 8. Page-level memory access frequency distribution.

4.3 Application Classification Process

Figure 7 illustrates the classification process where two tasks, JOB1 and JOB2, works together in SysMon. JOB1 is responsible for collecting the number of hot pages. Its sampling time interval is 3s in our system and the sampling duration in each interval is 10μs (at most). Our experiments show that 10μs is enough to collect sufficient information and incurs a negligible overhead. During each sampling JOB1 first clears *__access_bit* by the *pte_mkold()* kernel function, and then collects the number of hot pages (*__access_bit* set to 1) at the end of the sampling. Note that hot page numbers are averaged over several sampling intervals to avoid temporary spikes and reflect stable program behaviors.

JOB2 uses an array of page access counters to record the number of accesses for each page. Since the OS itself does not frequently reset the *__access_bit*, once set by the CPU, JOB2 employs a loop to periodically clear *__access_bit* and collects the access information during this period. JOB2 incurs slightly more overhead than JOB1, but the amortized overhead over the sampling time window (also 3s) is not high since it switches to the sleep mode after iterating 200 times (the time cost is far less than 3s). Based on the page access counters, JOB2 records the numbers of pages by grouping the counter values into five ranges: VH [150, 200], H [100, 150], M [64, 100], L [10, 64] and VL [1, 10]. For example, M denotes the number of pages with a counter value large than or equal to 64 but smaller than 100. By doing this, SysMon can gain a global view of the ***page-level memory access frequency distribution***. As illustrated in Figure 8 (a darker color indicates a larger number of pages), the distribution of pages with a different access frequency varies over different sampling intervals (horizontal axis). Due to the space constraint, Figure 8 shows only a small fraction of the sampling time period, which is representative of the whole period. Based on the above information, the WPD is computed as: ***WPD=(2×VH+1.5×H+1×M+0.5×L+0.1×VL)/all_used_pages***, where *all_used_pages* is the total number of pages accessed during a JOB2's sampling period (200 iterations). Notably, the row-buffer locality of one particular application can be estimated by WPD. Taking *sjeng* as an example, although it touches as many pages as some LLCM/LLCH applications, the per-page access frequency is relatively lower than that of a LLCM/LLCH application such as *mcf*. Correspondingly, the WPD value of *sjeng* is much lower than that of *mcf*.

In addition to the above metrics, ***page-level re-use time*** is also an important factor that can reflect an application's memory/cache behavior. In each sampling window, SysMon uses an array to record the number of intervals (i.e. the logic time) between two memory accesses to a specific page that cause *__access_bit* being set to 1. The pa-

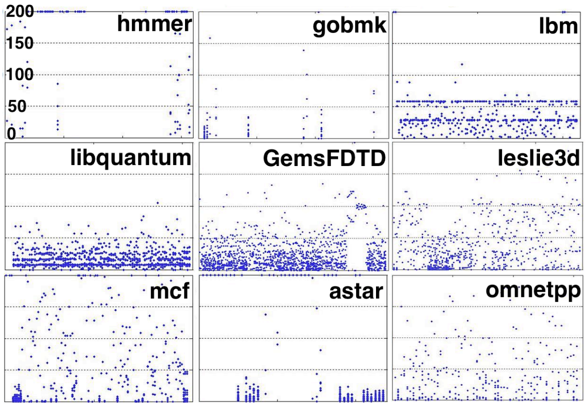


Figure 9. The re-use time of typical applications (the horizontal axis of each graph denotes the entire running time, and sysmon randomly selects one page in a sampling perild. The vertical axis denotes the logic re-use time ranging from 0~200 logic times).

-ge on which the re-use information is collected is randomly selected at the beginning of each sampling. This approach ensures that our collected re-use information is not biased and reflects the general trend over time. We tried several different random page selection mechinasms, and found the re-use patterns are highly similar. Figure 9 illustrates the re-use time sampling by SysMon across several representative benchmarks of different categories. An interesting finding is that LLCT applications often exhibit relatively more stable and lower re-use time. For example, *libquantum* and *GemsFDTD* show lower and more stable logical re-use time (around 20 on average), which are quite different from other types of applications. This feature can be used to identify LLCT applications effectively and efficiently in practice. Note that, in Figure 9, *lbm* exhibits a similar re-use time pattern observed in an LLCT application, but also behave in a similar manner as an LLCM application (performance drops modestly as cache capacity is reduced). For applications of this type, we found in our experiments that the LLCT feature often determines the overall performance. Thus, in our application classification algorithm we identify the LLCT feature first, before determining any other characteristics for an application.

Based on the WPD metric to measure reference locality, the hot page number and a series of thresholds, we devise a classification algorithm shown in Figure 7. The values of *rt_threshold (re-use time threshold)*, *ccf_threshold*, *hot_freq_threshold* and *llch_threshold* are 20, 100, 10% and 1000, respectively. The constants in our algorithm (i.e., sampling interval, weighted, thresholds, and etc.) are empirical values based on the analyses of all programs from SPECCPU 2006 with diverse memory features and a wide range of workloads combinations. Thus, we conclude that our approach would be cost-effective, robust and work well in real cases. These values can be adjusted as necessary as the environment or the workload changes.

5 Handling Multiplicity by Learning Rules

The application classification information obtained from the mechanism in Section 4 only reflects the partitioning preference for a single application (or applications with similar behaviors). The challenge of selecting an appropriate scheme for co-running applications with an arbitrary combination of memory demands remains unaddressed. To tackle this challenge, we adopt a data mining approach to quantitatively study the impacts of various memory allocation schemes on over 2000 workloads. We summarize the outcome as *partitioning/coalescing rules*, which can be used to handle diverse memory allocation needs for simultaneously running applications.

5.1 Partitioning Rules

Given a multi-programmed or multi-threaded workload, our first step is to select an appropriate memory allocation policy. To achieve this we collect a large amount of performance data from more than 10,000 experiments over 2000 workloads. For each workload, we use SysMon in Section 4 to obtain a classification vector, a notation to represent workload composition. For example, the classification vector of the workload {libquantum, mcf, bzip2, hmmer} is denoted as {<lib, LLCT>, <mcf, LLCH>, <bzip2, LLCM>, <hmmer, CCF>}. We run each workload with different policies and record the results as <cache-only: x%>, <bank-only: y%>, <A-VP: z%>, etc., where x%, y% and z% are performance improvements achieved by the corresponding policies. Based on the correlation between the classification vectors and the performance gains on different policies, we draw several interesting conclusions. First, almost all workloads that are combinations of LLCT and other type(s) of applications perform best on C-VP or A-VP. Second, a dominant percentage of workloads containing LLCH but not LLCT perform best on bank-only partitioning. Third, most workloads with LLCM but no LLCT or LLCH applications achieve best performance results with a modest cache partitioning scheme such as A-VP and B-VP. We summarize the above conclusions by the following three rules:

Rule-1: *Workloads containing LLCT and other applications (LLCH, LLCM, CCF) should use C-VP or A-VP* (*37.1% support, 94.4% confidence*[1]).

Rule-2: *Bank-only partitioning should be used for workloads with LLCH and LLCM applications but without LLCT applications* (*34.3% support, 83.3% confidence*).

Rule-3: *B-VP should be used for 8-programmed workloads with LLCM but no LLCT or LLCH applications* (*23.8% support, 87.9% confidence*).

The above analyses and rules also imply a priority in considering a memory allocation policy: LLCT > LLCH > LLCM > CCF. This ordering indicates that LLCT is the most "assaulting" type in that it brings a negative impact on virtually all the other types of applications while CCF is the most susceptible classification, and applications of this type hardly affect other applications' performance. These results can be also explained by architectural knowledge. In particular, Rule-1 is likely to perform well on any LRU (least recently used)-based caches since LLCT applications are not well handled by the LRU policy [10] as they waste other applications' resource without being

[1] *Confidence* and *support* are terminologies in data mining. In our work *support* is defined as the proportion of workloads that contain the specific types of applications in a rule; *confidence* indicates the accuracy of that rule.

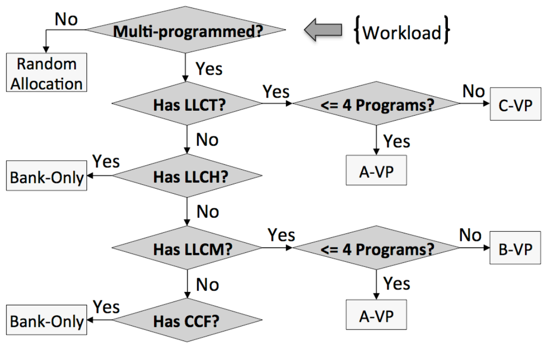


Figure 10. Memory allocation policy decision tree (PDT)

benefited. Rule-2 and Rule-3 can be also explained from the perspective of resource utilization.

For multi-threaded workloads, recent research [18] shows that bank partitioning only achieves a slight performance gain. Additionally, H. Park et al. [27] argues that a random page-interleaved allocation scheme outperforms partitioning schemes. We conducted experiments for multi-threaded (see Section 7) workloads and verified their conclusion. Thus, we add another rule for multi-threaded workloads:

Rule-4: *Multi-threaded workloads should use random page allocation.*

Based on the four rules and their priorities relative to each other, we generate a memory management **policy decision tree (PDT)** shown in Figure 10. The PDT is useful for choosing appropriate policies for diverse workloads.

5.2 Coalescing Rules and HVR

Despite the advantage in eliminating interference, a pure partitioning based approach (e.g., use rules 1~3) is not always preferable since it limits the cache capacity and can harm the performance for resource hungry applications (e.g., LLCH). Additionally, in real production environments, applications can be launched and terminated arbitrarily. This dynamically changing application environment can defeat any predetermined policy selection method based on a static knowledge of workload composition such as the static VP described in Figure 10. To arrive at a dynamic balance between partitioning and sharing, we extend the partitioning decision tree with several coalescing rules that can be used to merge the partitioned resource quotas among certain types of applications. We collect performance data of cache-only partitioning and represent the result for each workload as <(n×LLCT, m×LLCH, p×LLCM, q×CCF), *x%*>, where *n, m, p, q* are the numbers of applications of a certain type and *x%* is the performance gain achieved by the cache partitioning. Based on the results, we find that for almost all workloads that contain LLCH or LLCM but no LLCT applications ($n=0$, $m+p>0$) cache partitioning always hurts performance ($x\% < 0$). Additionally, for the workloads containing only LLCT applications ($m=p=q=0$, $n>1$), the improvement is quite modest ($x\%<1\%$) and for CCF dominant workloads ($n=m=p=0$, $q>1$) no obvious impacts are observed. Further, we run multiple LLCT applications together on 1/8 LLC capacity and find that the overall

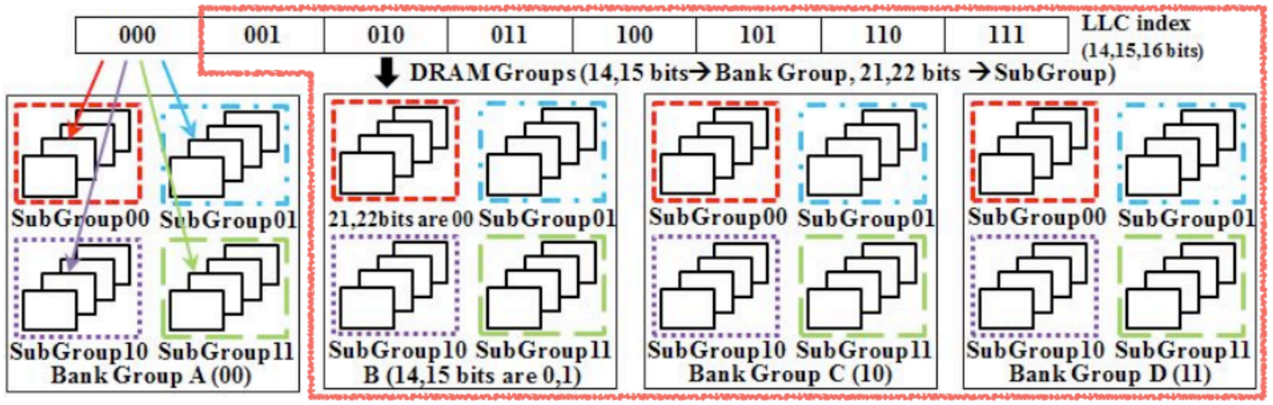


Figure 11. Cache/Bank allocation and memory mapping in Curve-VP (C-bit 16 and O-bits 15, 14 partition LLC into regions 000~111. Two cache regions share one bank group partitioned by bits 15, 14. B-bits 22, 21 further divide each bank group into four sub-groups). Colors are used to distinguish banks assigned to different threads. The figure is a "Memory-Map" to guide the allocation of Curve-VP.

performance is similar to the cases where they are partitioned or share the entire cache. The same results are observed for CCF workloads. Thus, we derive the following coalescing rules:

Rule-5: *LLCH and LLCM applications should be coalesced together to share the partitioned colors and cache quota* (*support*: *39.5%, confidence: 87.2%*).

Rule-6: *LLCT and CCF applications should be coalesced respectively to share the partitioned colors and small cache quota* (*support*: *7.8%, confidence: 90.5%*).

The coalescing rules are important complements to the partitioning rules for providing larger aggregate cache capacity and reducing misses under a partitioned cache. Utilizing partitioning and coalescing more flexibly is particularly useful for handling dynamic changes in the running workloads. For example, several partitioned cache quotas under C-VP can be dynamically coalesced to form a larger aggregate quota for accommodating multiple non-conflicting applications launched in an arbitrary order (see Rule-5 and Rule-6). On the other hand, a coalesced space can be partitioned if an additional partition is needed when an assaulting application (e.g., LLCT) is launched. By combining partitioning and coalescing, we develop a **HVR** (*Horizontal, Vertical and Random*) system, which adopts both partitioning and coalescing rules based on the composition of a workload (extracted from SysMon). HVR starts with bank-only partitioning upon kernel boot (to bring stable performance gains for all applications and reduce the amount of pages need to be migrated when the policy changes). Thus, all running application(s) share the entire cache capacity initially. As more applications are launched and categorized by SysMon, HVR partitions or merges cache space based on the aforementioned partitioning and coalescing rules.

5.3 Curve-VP (Curve-Vertical Partitioning)

The baseline HVR system introduced in Section 5.2 has the capability of handling dynamic workloads, but may incur a page re-coloring and migration overhead to reach a stable state. The page migration overhead can be non-negligible in cases where frequent partitioning and coalescing happen upon workload changes. For short-running and transient application loads, starting with an all-shared approach (bank-only partitioning) and adapting to other policies after training period might incur relatively expensive costs. To avoid the overhead incurred during partitioning and coalescing in the baseline HVR, we introduce a special type of partitioning technique: *curve vertical partitioning (**Curve-VP**)*. To reduce page migration, Curve-VP features a relatively "rigid" yet effective partitioning approach: allocate a small slice of cache quota (1/8) to all LLCT applications while having all the other types of applications to share the remaining cache space. Although merging CCF with LLCH and LLCM is conflict with the partitioning rule, we found that with modern multi-level caches CCF applications can be largely satisfied by upper cache levels (L1 and L2), thus their LLC quota can be merged to share the same region with LLCH and LLCM applications without significant performance penalty (see Section 8 for more details). We also found that reserving a small cache region for CCF will degrade LLCM and LLCH performance, which in some cases cannot be compensated by the benefit in CCF applications.

As opposed to the baseline VP, Curve-VP is initialized to use a heavily partitioned strategy. In particular, any process, upon creation, is assigned the cache region (denoted by 16, 15 and 14 bits) 000 (by default, but this may change based on real cases and user's demands). After SysMon collects enough information for this process (often after several sampling windows in seconds), the system determines the category of the process. If the process is identified as an LLCT application, no change is needed and the subsequent page requests from this process still uses the same dedicated cache region (Figure 11) to avoid thrashing effects for the other types of applications. Although in an extreme case a number of LLCT applications can be squeezed into a small cache quota, their performance is not likely to be greatly impacted due to their poor cache utilization. Additionally, the number of simultaneously running LLCT applications is typically limited by the number of cores, thus cannot pose a high pressure to cache. DRAM banks, as an independent dimension, can still be partitioned using the higher order bits (22, 21) to eliminate interference at bank level. In particular, each LLCT application can use one or more sub-

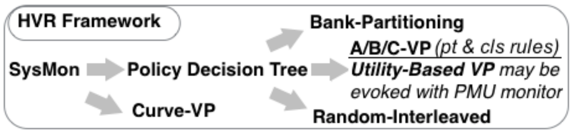


Figure 12. HVR Framework in a nutshell (with Curve-VP). Note that Curve-VP can be replaced by bank partitioning and Random-Interleaved policies.

groups in the bank group A (colors are used to distinct different threads), dictated by the indexing bits representing cache region 000. On the other hand, if the process is classified as LLCH, LLCM, or CCF, the subsequent page requests are directed to region 001~111. The original pages left on region 000 are migrated lazily when they are accessed again, thus minimizing the migration cost (only hot pages need to be migrated). Note that although these applications share a large cache region (001~111), the memory accesses are isolated at DRAM bank level. Marked by the big red rectangle in Figure 11, in Bank Group B, C and D, each SubGroup (denoted by different colors and indexed by bits 21,22) is used by an application. Moreover, since Curve-VP has the knowledge of "Memory-Map" (Figure 11) and the applications' memory features, it can avoid the unnecessary inter-thread memory conflicts and the overhead caused by page migration. The allocation process works as an expanding "balloon" at both LLC and DRAM level.

5.4 Combining Policies into HVR Framework

Our final HVR framework includes A/B/C-VP (with partitioning and coalescing rules in dynamic cases), Curve-VP, bank partitioning and Random-Interleaved allocation (illustrated in Figure 12), forming a flexible and large design space. Note that Utility-based VP[2] can also be intergraded into HVR by employing PMU hardware counters. Curve-VP is particularly useful in a highly dynamic application environment that would result in a large amount of page migration if dynamic VP were used. Curve-VP gives the largest possible amount of cache space for applications that need cache resource while limiting the resource for those with poor cache utilization. At the DRAM bank level, partitioning is still enabled to further segregate the potentially interfering applications that share the same cache space. HVR can be adopted on a wide spectrum of hardware and architectures, as O-bits are common in many commercial platforms. Although there are only two O-bits in our experimental systems, studies show that more O-bits can potentially benefit the overall system performance and bring more flexibility and optimization opportunities.

6 X-BUDDY: SUPPORTING HVR IN LINUX KERNEL

The previously presented classification framework is implemented as kernel modules in the Linux kernel 2.6.32 in about 400 lines of source code. This section details our modifications to the kernel data structures and paging algorithm to support the previously discussed policies and the partitioning/coalescing rules in a unified system. We implement HVR in roughly 2000 lines of source code over the existing kernel source tree. HVR was originally supported by two page-indexing systems: sub-system A and sub-system B [20]. To simplify the page indexing system and reduce the implementation overhead, we merge sub-system A and sub-system B into one indexing system named x-Buddy, as illustrated in Figure 13.

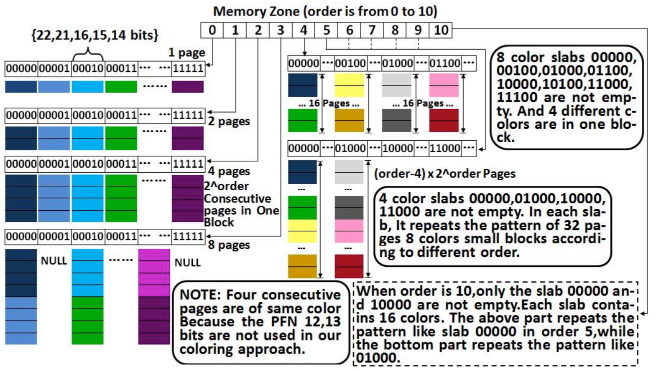

Memory Zone (order is from 0 to 10)
0 1 2 3 4 5 6 7 8 9 10
{22,21,16,15,14 bits}
1 page
00000 00001 00010 00011 ··· ··· 11111
2 pages
4 pages
2^order Consecutive pages in One Block
8 pages
NULL
NULL
NOTE: Four consecutive pages are of same color Because the PFN 12,13 bits are not used in our coloring approach.
00000 ··· 00100 ··· 01000 ··· 01100 ···
... 16 Pages ...
8 color slabs 00000, 00100,01000,01100, 10000,10100,11000, 11100 are not empty. And 4 different colors are in one block.
00000 ··· 01000 ··· 10000 ··· 11000 ···
(order-4) x 2^order Pages
4 color slabs 00000,01000,10000, 11000 are not empty. In each slab, It repeats the pattern of 32 pages 8 colors small blocks according to different order.
When order is 10,only the slab 00000 and 10000 are not empty.Each slab contains 16 colors. The above part repeats the pattern like slab 00000 in order 5,while the bottom part repeats the pattern like 01000.

Figure 13. x-Buddy system. Organized by O-bits 14,15; C-bits 16 and B-bits 21,22. It supports (Curve-) VP, bank partitioning and Random-Interleavied allocation policies.

As in conventional Linux kernel, x-Buddy system maintains free physical pages in orders of blocks from 0 to 10. Each block in order-*n* contains 2^N continuous pages. The five bits (22, 21, 16, 15, 14) from page frame number (PFN) used in x-Buddy system form a set of 32 colors (00000~11111), each of which has a free page list in our modified kernel. As the selected five coloring bits cover all the three categories (i.e., bits 22 and 21 are B-bits, bits 15 and 14 are O-bits, and bit 16 is C-bit), x-Buddy system can support A/B/C-VP and horizontal bank partitioning by choosing different coloring policies. In our platform, the page offset is from bit 0 to 11 and the PFN starts from bit 12. As the color bits begin from bit 14, every four consecutive pages in physical address share the same color. In order-0 (upper left corner of Figure 13), each block is an individual page and the block list under a particular color is a set of pages of that color. For example, the block list under green color in order-0 contains any free and non-continuous pages with the five coloring bits being 00011. Order-1 and order-2 are similar to order-0 except that two and four pages in a block are continuous. Each block in order-3 has eight continuous pages and thus spans two colors since there is one coloring bit (bit 14) within the offset of a block in order-3. Similarly, a block in order-4 spans two O-bits (bits 15 and 14) and thus has four colors (16 pages), as depicted in Figure 13.

All the policies in our HVR framework can be supported in x-Buddy system. In particular, x-Buddy supports A-VP as the page coloring bits include bits 14 and 15. From the perspective of A-VP, the bit 16, 21 and 22 are not coloring bits thus the buddy system views two colors with the same lower two bits to be the same color (e.g. 00100 and 01000), or a color group. Each color group in the four color-group pool dictated by bits 14~15 (00, 01, 10 and 11) represents one quota in A-VP, which partitions

[2] Utility-based VP (UVP) starts with A/B/C-VP and dynamically adjusts cache partitioning based on cache misses monitored through hardware performance counters. It is a normal approach with no pt&cls rules, and may incur bank conflicts because it has no knowledge of "Memory-Map". We use Utility-Based approach as one of the base line in our following experiments.

both cache and bank into four slices. In Figure 13, eight colors with the same value at bit 14~15 form one color group for A-VP (named an A-VP color group). For one particular application, allocating pages within a dedicated A-VP color group ensures the pages used by the application are partitioned or segregated from other applications using A-VP and thus the application only uses one quota in A-VP. Similarly, x-Buddy supports C-VP by considering page coloring bits 14 ~16. Each color in the eight color group pool dictated by bits 14~16 (000~111) represents one quota in C-VP. By applying the same principle, x-Buddy supports B-VP by leveraging B-bits (21,22) and O-bits (14,15), which can be used together to partition cache into up to four partitions and bank into up to 16 partitions. Therefore, a page requested by any application can be handled in x-Buddy using A/B/C-VPs or horizontal partitioning policies.

In x-Buddy system, the random-interleaved page allocation for multi-threaded workloads can be achieved by randomly selecting pages in the order-0 free list. Moreover, similar effect achieved by M^3 [27] can be also supported in x-Buddy by allocating physical pages with all 32 colors (00000~11111) in a round-robin fashion and interleaving requested pages evenly across all banks to reduce the potential bank conflicts. Based on the x-Buddy system, we develop a hash-based searching algorithm (see Pseudocode 1) to allocate a page in O(1) time (O(logn) time complexity in extreme cases, where memory blocks are frequently merging and splitting).

Pseudocode 1: Hashing algorithm for selecting pages
Input: (1) order; (2) target_color **Output:** one page of target color

```
BEGIN
/*CASE: Physical pages organized based on bits 14~ 16, 21~22*/
IF using 14, 15,16, 21, 22 THEN
  SWITCH (order)
```

case	0~2	3	4	5~9	10
colors_per_block	1	2	4	8	16

```
  END SWITCH
  block_color = (target_color / colors_per_block) ×colors_per_block;
  //The 4th bit (21 bit) is 1
  IF order is 10 AND the color bits are x1xxx THEN
      page_index = (target_color - block_color - 8) × 4 + (1 << 9);
  ELSE  page_index = (target_color - block_color) × 4;
  END IF
END IF
Expand color block (page_index, order)
RETURN page[page_index] and remove this page from free list.
END
```

* target_color is the color of the requested page.
* block_color is the color of the first page in a block.
* colors_per_block is the number of colors in a block.

7 Evaluations

7.1 Experimental Methodology

Our experimental machine has a quad-core eight-thread 2.8GHz Intel i7-860 processor with 8MB 16-way LLC and 8GB 64-bank (125MB/bank) DDR3 main memory. The machine runs CentOS Linux 5.4 with the kernel 2.6.32. We use SPECCPU2006 suite [1] for multi-programmed workloads and PARSEC 2.0 benchmark suite [2] for multi-threaded workloads. All programs are compiled by gcc 4.4.3 with the O3 optimizations. We use *weighted speedup* [13] (WS) to measure system performance and *maximum slowdown* [13] for fairness. To evaluate our approaches, we randomly generate tens of workload combinations of different types of applications, and measure their performance on different polices (Utility-based cache and bank partitioning[3], Curve-VP). For each combination we run the experiment several times to average out the variations. The baseline is un-modified buddy system in Linux kernel 2.6.32.

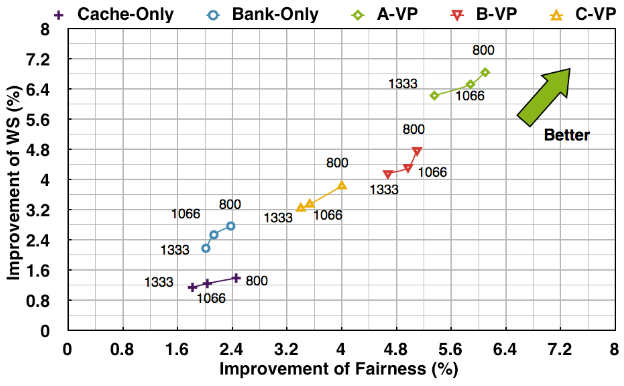


Figure 14. Performance of different policies as bandwidth changes (the baseline is the unmodified Linux kernel).

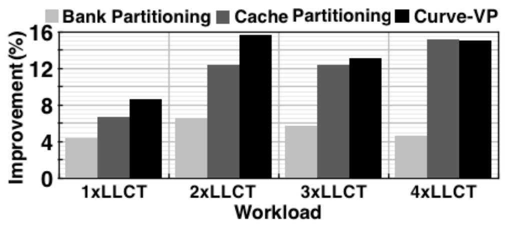


Figure 15. Average performance improvement of the bank partitioning, cache partitioning and Curve-VP with different number of LLCT applications. Enabling coalescing in cache partitioning in some cases for optimal performance.

7.2 The Effectiveness of VP

In our experiments, we test a large amount of workloads to show the effectiveness of VP and its advantages against horizontal partitioning. Figure 14 summarizes the performance and fairness improvements of various policies based on an average of workload performance in our experiments. To demonstrate that the proposed scheme performs robustly under different memory bandwidth, we change the memory frequency from 1333 to 800MHz. Figure 14 illustrates that on average all the three vertical partitioning policies outperform the horizontal cache-only or bank-only partitioning schemes. Particularly, A-VP is nearly 6% better than cache-only partitioning and 5% better than bank-only partitioning. As bandwidth decreases, the contention becomes more severe and the three vertical policies can bring even larger improvements. Therefore, we can draw conclusion that vertical partitioning brings additional benefits over horizontal partitioning and is a promising memory management mechanism for future multicore systems with increasing bandwidth pressure.

[3] We have tried the optimal (with coalescing enabled in some cases) utility-based (using PMU) dynamic cache/bank partitioning approaches in our experiments, and compare them with our work.

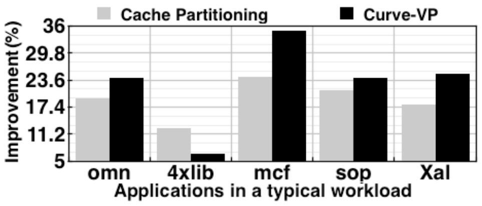


Figure 16. The performance trade-off for different applications in one workload with 4 LLCT applications.

However, we find that for some LLCT dominated workloads, VP performs similar to the optimal cache partitioning. To further study this, we conduct experiments using tens of workloads that contain different numbers (1~4) of LLCT applications, and each of them also contains several LLCH or LLCM applications. We use the newly proposed Curve-VP, since it often incurs low overhead and brings optimal performance. Figure 15 presents the average performance for these workloads on three representative partitioning schemes. As the leftmost bars show, with one LLCT application, all the three partitioning schemes bring certain amount of performance improvement. Curve-VP performs better than bank partitioning and cache partitioning. With two LLCT applications, the thrashing effect becomes more severe in the baseline system and all three partitioning schemes bring much higher performance gains than the one-LLCT case. Curve-VP performs the best, bringing a nearly 16% improvement over the baseline where LLCT applications significantly interfer with LLCH/LLCM applications, and around 4% over cache partitioning on 20 random workloads. With three or four LLCT applications, Curve-VP still brings great benefits, however it performs similarly to the cache partitioning**.**

This result indicates that the performance of Curve-VP relative to the cache partitioning in these cases (3, 4 or more LLCT applications) is limited due to the interference among LLCT applications and the resource constraint (more LLCT applications are mapped into **Bank Group A** in Figure 11). Although the performance gains on the cache partitioning and Curve-VP are similar with high number of LLCT applications, segregating LLCT applications brings an opportunity to trade the performance among applications. Figure 16 illustrates the performance for each application in such a typical workload (4×LLCT + 4×LLCH or LLCM). As can be observed in Figure 16, for LLCH and LLCM applications (*omn, mcf, sop, Xal*) Curve-VP outperforms the cache partitioning approach**,** but the four LLCT applications show a worse performance on Curve-VP than on the cache partitioning. Clearly, when the number of LLCT applications is high, the performance slowdown caused by LLCT applications can offset a considerable portion of the benefits brought by LLCH or LLCM applications. This inspires us that VP can be used to balance the performance across applications, and trade resource-sensitive applications's performance for QoS.

7.3 Overall Performance Comparison

To demonstrate the superiority of HVR framework, we conduct experiments to benchmark its component policies (i.e. static A/B/C-VP and the corresponding dynamic Utility-based VP, Curve-VP), and its capability of selecting proper policy and enabling the coalescing and partitioning rules. Static vertical partitioning (SVP) adopts A-VP for 4-programmed workloads and B/C-VP for 8-programmed workloads based on the PDT tree. Utility-based VP (UVP) dynamically adjusts cache partitioning based on cache misses monitored through hardware performance counters. HVR is our comprehensive framework that combines bank and vertical partitioning and coalescing rules based on the workload composition from SysMon. HVR enables Curve-VP when needed.

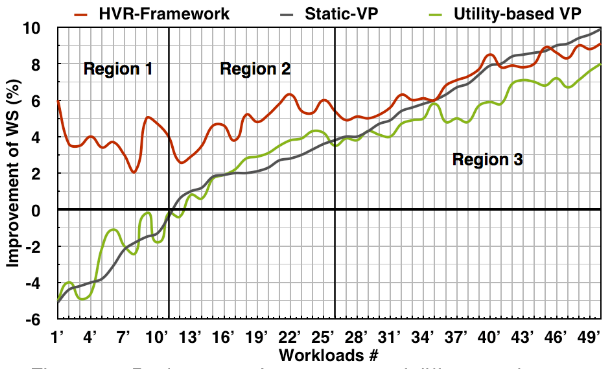


Figure 17. Performance Improvement of different schemes.

7.3.1 Dynamic Policy Selection of HVR

Figure 17 reports the performance for the three schemes over 50 randomly generated workloads sorted by their performance improvements achieved by SVP. In region 1, both SVP and UVP achieve negative performance gain (up to -5.0%) over the baseline. In contrast, HVR improves performance by up to 6.1% over the baseline and 11% over SVP and UVP. A careful analysis reveals that workloads in region 1 are primarily LLCH dominated workloads, for which the cache partitioning is detrimental. Thus, SVP and UVP policies are ill suited for these workloads. HVR achieves gains by automatically identifying the workload characteristics and performing the bank-only partitioning for them.

In region 2, most workloads are 8-programmed ones with LLCT applications. HVR outperforms SVP and UVP due to resource coalescing. For instance, the workload 22′ contains 5 LLCT, 2 LLCH and 1 LLCM applications. HVR maps all LLCT applications to 1/8 cache, leaving the remaining 7/8 cache shared by LLCM and LLCH applications. The mixed partitioning and sharing improves the performance in region 2. In most cases of region 3, HVR outperforms other two approaches, since HVR is capable of selecting proper VP policies and using coalescing rules. But for some workloads, SVP performs slightly better (0.4% better than HVR on average). Looking into the workloads in this region we find a high percentage of A/B-VP friendly workloads containing multiple LLCM applications. Since an LLCM application requires modest cache capacity and typically maintains a steady rate of cache utilization. SVP is effective as it uses offline profiling and partitions the cache at the beginning. It avoids the dynamic overhead of an online method. In comparison, dynamic utility-based approaches incur a non-negligible overhead [17,39] due to expensive page migration induced by page re-coloring and performance counter penalty. Fortunately, HVR avoids offline profiling

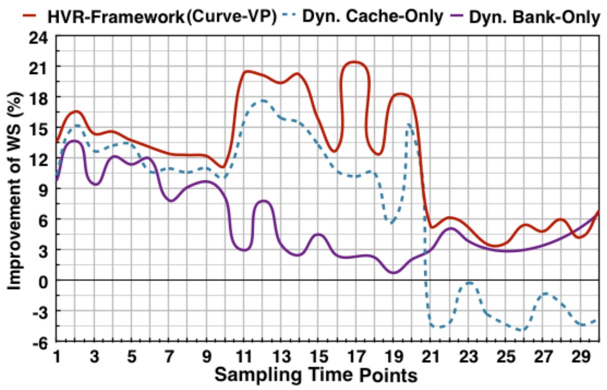


Figure 18. Real-time performance of HVR (Curve-VP enabled).

and does not incur significant overhead due to the page-table-based lightweight online profiling and the stable classification approach (more details are in section 8).

7.3.2 Real-time Performance for HVR (Curve-VP)
As previously mentioned, Curve-VP is more suitable for a dynamically changing workload, where new jobs may be submitted for execution and existing jobs may terminate at any time. Figure 18 reports the real-time performance for cache-only partitioning, bank partitioning and HVR (with Curve-VP enabled), measured in IPC using Intel processor's performance counters.

During testing, we inject applications of various categories at random time points. Previously launched application can stop running during the testing period. To make a fair comparison, the injection time points, the application input parameters and the input sequence are the same for all three schemes as well as for the baseline. From the figure, we can see the performance of cache partitioning fluctuates between 18% and -5%.

At the sampled time points 21~30, the performance gain of cache partitioning drops below zero. This is because at these points LLCH applications are launched, and the performance degrades due to a mixed effect of limited cache resources and bank-level contention. HVR (with Curve-VP enabled) can avoid this degradation by mapping the LLCH applications to isolated sub-bank groups and allowing them to use a large cache space (See Section 5). At the sampled time point 17, HVR achieves the peak IPC improvement. This is the point where 2 LLCT and 4 LLCH/LLCM applications are running together. Curve-VP achieves the best performance among the three schemes, by mapping LLCT applications into small number of cache sets, allocating a larger cache space for LLCH/LLCM applications, and isolating all applications at DRAM bank level. Curve-VP incurs little overhead as it directly uses the pre-determined optimal mapping approach, and the page migrations only involve the hot pages.

Compared to Curve-VP, bank/cache-only partitioning approaches achieve modest gains (5% and 7% worse than HVR, respectively) due to the inability to vary policies and coalesce resources dynamically, and the high resource migration overhead. Note that bank-only partitioning achieves relatively stable performance over time compared with cache-only partitioning and this trend is consistent with our conclusion in Section 3 (see Figure 3).

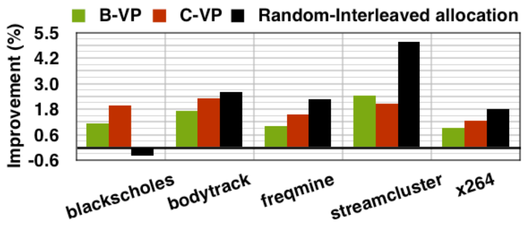


Figure 19. Performance of multi-threaded workloads.

7.3.3 Performance for Multi-threaded Applications
As previously mentioned, performance benefits can be achieved for multi-threaded workloads by adopting a random, interleaved page allocation approach. In Figure 19, we show the random-interleaved page allocation policy outperforms B/C-VP policies for various 8-threaded workloads. HVR supports random-interleaved page allocation (see Section 5, Section 6.2, and Figure 7), which is automatically selected for a multi-threaded workload based on the policy decision tree (PDT).

8 Discussion

(1) Overhead of HVR. Overhead of HVR comes from the following three sources: **1).** Page table sampling of JOB1 and JOB2 in the workload classification process. The costs of page table traversal depend on an application's memory footprint. In our experiments, the time for page table traversal ranges from 5µs (*povray*) to 4.46ms (*mcf*). Thus, the amortized overhead of JOB1 and JOB2 are negligible. Moreover, JOB2's sampling interval grows with an increasing step once it collects sufficient information to complete the initial classification process, and thus its overhead is further reduced for long running workloads. In the worst case, JOB2 only adds 0.6% overhead. For workloads with an extremely large memory footprint, random sampling can be adopted for a tradeoff between the sampling overhead and classification accuracy. **2).** The page indexing in the modified buddy system. As our page searching routine can allocate a page in O(1) time in common cases, x-buddy system incurs a negligible overhead (< 0.1% on average) during page allocation. **3).** Page migrations caused by re-coloring in dynamic policy adjustment. Migrating a 4KB page costs around 3µs on our platform. Fortunately, VP does not incur too many page migrations because it relies on stable classification information that typically changes only when an application starts or terminates. Moreover, since our mechanism uses the lazy page migration [17] that only migrates a page when necessary (the "hot" ones), the average overhead is less than 0.8% (1.7% at most in one extreme case). HVR performs better than the previous Utility-based approaches, in which the number of migrated pages fluctuates over time and incurs a higher migration overhead in practice. In Curve-VP, the page migration overhead is the lowest with the dynamically changing workload due to the fact that the allocator has the full knowledge of the memory mapping and uses the pre-determined, highly efficient, "ballooning" allocation. In our future work, we will try DMA to further reduce the overhead.

(2) Using Dedicated Region for CCF applications? Our results indicate that CCF applications should use dedicated region in certain cases, but share the same space with LLCH and LLCM applications in other cases. For 4-programmed workloads with/without LLCT applications, we found that, on average, the performance degradation (compared to single thread run) for CCF applications is only 1~3%, compared to 5~10% (w/o LLCT) and 10~40% (w/LLCT) for LLCH/LLCM applications. In other words, using a shared region for CCF, LLCM, and LLCH applications does not harm CCF performance significantly. Thus, allow LLCM and LLCH to use a larger space together with CCF increases more performance benefits. However, if more applications are running (e.g., 8-programmed workload) and CCF applications dominate a workload (more than half of the applications are CCF in the workload), it is beneficial to segregate CCF from other types of applications. This is because the accumulated performance lost seen by CCF applications (CCF may suffers 10~30% performance lost on average) may offset the benefits brought by cache sharing among LLCH, LLCM and CCF. Our HVR framework can detect this case and segregate CCF applications as needed. Additionally, HVR provides interfaces to users to tune its performance by using empirical parameters, experiences, and offline knowledge. **(3) Industry Impact and Future Direction.** The benefits of vertical memory management to industrial world are multifold. **1).** It adds both cache and DRAM into the OS management pool, and thus potentially benefits the overall system performance by simultaneously reducing cache and DRAM contention, a critical problem faced by cloud providers such as Amazon, Google and VMware. **2).** It significantly enlarges the memory management policy space and brings greater flexibility for diverse application needs in commercial data center and production environments. Moreover, application memory access and usage patterns are captured using a practical, page-table sampling. **3).** It should also reduce the energy cost and access latency of emerging non-volatile memories (NVMs). In particular, memory-partitioning techniques are useful for NVMs where row buffer miss latency and energy cost is high. **4).** It segregates applications with high latency-sensitivity from those with high bandwidth-sensitivity (e.g., streaming applications), thus ensuring better QoS and fairness via throtting the resource utilization. **5).** Our partitioning and coalescing techniques can be used together to handle dynamic workload changes in production environments, thus having a broad influence on efficient resource isolation, virtualization and consolidation, which are critical and have a significant impact on the trend of "moving to the cloud". **6).** Our work demonstrates that applications, systems, architectures and hardware can be more tightly coupled together for a more flexible design and optimization space. It reveals the trend that the hardware and software vendors are working closer with each other to pave the way of delivering truly software-defined, application-aware computer system and hardware.

By restructuring the buddy system slightly, we implement the HVR framework as an all-in-one solution that combines horizontal, vertical partitioning and random allocation. We believe our prototype demonstrates the feasibility of a more intelligent memory management strategy in modern OS design for addressing the emerging challenges in future complicated computing environments. It requires only a minimal effort to port our prototype to production settings to support diverse commercial workloads.

9 Related Work

There is a large body of related work on cache and memory allocation and partitioning. At the main memory level, Prashanth et al. [26] proposes DRAM channel partitioning that requires hardware and system modifications to segregate data from different threads into different channels to eliminate interference. Park et al. [27] adopt a random allocation algorithm to scatter allocated pages to multiple banks to avoid conflicts for multithreaded workloads. Liu et al. [18-20] use page-coloring to partition DRAM banks/Channels to avoid contention of multiple programs. Kaseridis et al. [12] propose bandwidth-aware memory sub-system management for avoiding resource contention. Various approaches are also proposed to manage LLC [6,9,15-17,21,28,30,32,38]. In particular, Qureshi et al. [28] design a utility-based cache partitioning scheme that allocates appropriate cache resources based on application miss rate monitored through dedicated hardware. More recently, cache partitioning is also adopted in heterogeneous GPU-CPU architectures to promote fair resource sharing among CPU and GPU applications [22,25], which exhibit different memory access characteristics. Many efforts [4,10,13,17,24,36,41] classify workloads based on hardware profiling, and then choose appropriate scheduling policies for different classifications or create performance model for analysis. The latest work in [3] proposes a promising cache partitioning and sharing approach based on a recent advance in locality theory in [37]. OS-level approaches for memory utilization monitoring [7,8,39,40] have also been studied to assist resource management.

10 Conclusions

In this paper, we try to rethink and answer the question about best-fit memory allocation approach in modern OS. We propose and implement a practical, unified, and efficient multi-policy memory management framework named HVR to address the challenge of allocating appropriate memory resources for modern diverse applications. HVR seamlessly integrates several existing schemes and new vertical partitioning policies by leveraging O-bits and the page-coloring technique. Through a quantitative study on a large quantity of experiments we verify that HVR can automatically select appropriate policies based on application needs and achieve 20% performance benefits compared to prior allocation methods in many cases.

Acknowledgment

We thank the reviewers for their feedbacks. Lei Liu, Hao Yang and Chengyong Wu are supported by the 863 Program under grant No. 2012AA01092, 2012AA010901 and

973 Program under grant No. 2011CB302500; the National Natural Science Foundation (NSF) of China under grants No. 61303051, 61202055, 60921002, 61303052, 61303053 and 61402445. Chen Ding is supported by NSF under grant No. CNS-1319617, CCF-1116104, CCF-0963759 and NSFC No. 61232008 and an IBM CAS Faculty Fellowship. We would like to extend our deep thanks to Prof. P.C.Yew and Xiaodong Zhang for their valuable comments. We also thank these authors who pay attentions to the prior version of this study, especially Zehan Cui.

REFERENCES

[1] Standard Performance Evaluation Corporation. Available from: http://www.spec.org/cpu2006/CINT2006/.
[2] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In PACT. 2008.
[3] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal Cache Partition-Sharing. In ICPP. 2015.
[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In HPCA. 2005.
[5] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In MICRO. 2006.
[6] H. Cook, M. Moreto, S. Bird, K. Dao, D.A. Patterson, and K. Asanovic, A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In ISCA. 2013
[7] P. J. Denning, The Working Set Model for Program Behaviour. In Commun. ACM, 1968. 11(5).
[8] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. In EuroSys. 2011.
[9] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In PPoPP. 2011.
[10] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer, CRUISE: cache replacement and utility-aware scheduling. In ASPLOS.2012.
[11] M. K. Jeong, D.H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In HPCA. 2012.
[12] D. Kaseridis, J. Stuecheli, J. Chen, and L.K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In HPCA. 2010.
[13] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In MICRO. 2010.
[14] K. C. Knowlton, A Fast storage allocator. In Communications of the ACM, 1996.
[15] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In PACT. 2004.
[16] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In RTAS. 1997.
[17] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In HPCA. 2008.
[18] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In PACT. 2012.
[19] L. Liu, Z. Cui, Y. Li, et al. BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems. In TACO. 2014.
[20] L. Liu, Y. Li, Z. Cui, et al. Going Vertical in Memory Management: Handling Multipolicity by Multi-Policy. In ISCA. 2014.
[21] Y. Li, R. Melhem, A. K. Jones. Practically Private: Enabling High Performance CMPs Through Comiler-assisted Data Classification. In PACT. 2012.
[22] J. Lee and H. Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In HPCA. 2012.
[23] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. In NPC. 2010.
[24] L. Ma, K. Agrawal, R D. Chamberlain. A Memory Access Model for Highly-threaded Many-core Architecture. In Future Generation Computer Systems. 2014.
[25] L. Ma, K. Agrawal, R. D. Chamberlain. Theoretical Analysis of Classic Algorithms on Highly-threaded Many-core GPUs. In Proceedings of 19rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Feb 2014.
[26] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In MICRO. 2011.
[27] H. Park, S. Baek, J. Choi, D. Lee, and S.H. Noh, Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems, In ASPLOS. 2013.
[28] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In MICRO. 2006.
[29] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. In ISCA. 2000.
[30] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In ICS. 1999.
[31] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In MICRO. 2008.
[32] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. In MICRO. 2009.
[33] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In WIOSCA. 2007.
[34] A. S. Tanenbaum, Modern Operating Systems. 3rd ed. 2008: Pearson-Prentice Hall.
[35] A. Wolfe. Software-based cache partitioning for real-time applications. In RCS. 1993.
[36] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects. 2008.
[37] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a Higher Order Theory of Locality. In ASPLOS. 2013.
[38] Y. Xie and G. H. Loh. Scalable shared-cache management by containing thrashing workloads. In HiPEAC. 2010.
[39] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In EuroSys. 2009.
[40] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In ASPLOS. 2004.
[41] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In ASPLOS. 2010.

Lei Liu received the Ph.D in Advanced Compilation System Group in State Key Laboratory of Computer Architecture (SKL), Institute of Computing Technology (ICT), and Chinese Academy of Science. Before He joined ICT, he spent several years in industry after receiving MS degree in University of Science and Technology of China, and BS degree in Dalian University and Technology. His research interests are in high performance memory system, OS design, and the optimization/evaluation for modern computer architectures. As a leading author, his efforts are published in ISCA, PACT, TACO and IEEE TC. He is a member of the IEEE Computer Society.

Yong Li received the PhD degree in computer engineering from University of Pittsburgh in 2013. His research interests include highperformance computer architectures, compilers, and parallel systems. He is currently a Member of Technical Staff at VMware PA. CA, US, and some of his efforts are published in ISCA, PACT, ISLPED, TACO, IEEE TC and TPDS, etc.

Chen Ding is a Full Professor at University of Rochester, USA. His research includes locality theory and optimization and has received young investigator awards from NSF and DOE. He co-founded the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPC).

Hao Yang graduated from Nanjing University of Technology with BS degree in Computer Science. He is a research assistant in SKL under the supervision of Prof. Chenggang Wu and Lei Liu. His research interestings are in "Computing in Memory", and modern OS design.

Chengyong Wu was a Full Professor in ICT, CAS. His research spans instruction-level parallelizing compilation, parallel programming model and language, and iterative and adaptive optimization. He has served as PC member of several international conferences like PLDI, CGO, ICPP, HPCC, etc, and some of his efforts are published in ISCA, ASPLOS, PLDI, PACT, TACO, FPGA, etc.