2017 IEEE International Symposium on High Performance Computer Architecture

SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support

Xiaodong Wang Shuang Chen Jeff Setter[†] José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
http://m3.csl.cornell.edu/

[†]Dept. of Electrical Engineering
Stanford University
Stanford, CA 94305 USA

***Abstract*—Performance isolation is an important goal in server-class environments. Partitioning the last-level cache of a chip multiprocessor (CMP) across co-running applications has proven useful in this regard. Two popular approaches are *(a)* hardware support for way partitioning, or *(b)* operating system support for set partitioning through page coloring. Unfortunately, neither approach by itself is scalable beyond a handful of cores without incurring in significant performance overheads.**

We propose SWAP, a scalable and fine-grained cache management technique that seamlessly combines set and way partitioning. By cooperatively managing cache ways and sets, SWAP ("Set and WAy Partitioning") can successfully provide hundreds of fine-grained cache partitions for the manycore era.

SWAP requires no additional hardware beyond way partitioning. In fact, SWAP can be readily implemented in existing commercial servers whose processors do support hardware way partitioning. In this paper, we prototype SWAP on a 48-core Cavium ThunderX platform running Linux, and we show average speedups over no cache partitioning that are twice as large as those attained with ThunderX's hardware way partitioning alone.

I. Introduction

Performance isolation is an important goal in server-class environments for a variety of reasons, including throughput, quality of service, and even security. Partitioning last-level caches in chip multiprocessors (CMPs) across applications is a popular approach to reducing or eliminating interference across applications co-running on a CMP. It is a mechanism that can help (1) maximize resource utilization and system throughput, or trade off throughput vs. fairness [42], [43]; (2) provide quality-of-service (QoS) for latency critical workloads [25]; (3) protect the system from timing channel attacks, where a malicious program is able to steal the secure information of another application, such as the encryption key, by sharing the last-level cache [5]. A few approaches to partitioning the cache space have been proposed.

Way partitioning allows cores in chip multiprocessors (CMPs) to divvy up the last-level cache's space, where each core is allowed to insert cache lines to only a subset of the cache ways. It is a commonly proposed approach to curbing cache interference across applications in chip multiprocessors (CMPs) [30]. Unfortunately, way partitioning is proving to be not particularly scalable, as it affects cache latency and power negatively, eventually becoming impractical. Consider that multiple current and upcoming server chip multiprocessor (CMP) lines already comprise twenty, thirty, or even more cores; examples include Intel's 22-core E5-2600 v4, IBM's 24-core Power-9, Cavium's 48-core ThunderX, or Qualcomm's 64-core Hydra. Although some of these processors do include hardware support for way partitioning, the granularity is too coarse to allow for separate partitions for more than a handful of applications. Cavium's ThunderX processor, for example, possesses 48 cores, however its last-level cache is limited to "only" 16 ways. Similarly, Intel's v4 CMP allows for no more than 20 different partitions across 22 cores.

Another approach to achieving cache partitioning is to restrict each application's page frames to certain "colors" (the shared bits between a physical address' page frame ID and cache index). In this case, page frames of each color map onto a specific subset of the cache sets. Although this approach has been adopted in real operating systems [22], [24], [46], it also does not scale beyond a handful of colors.

A few architectural mechanisms for probabilistic fine-grain cache partitioning have been proposed [26], [34], [41]. However, these implementations require extra hardware support, do not provide true isolation, and have not yet been adopted in any commercial CMP to our knowledge.

Contributions

We propose SWAP, a fine-grained cache partitioning mechanism that can be readily implemented in existing CMP systems. By cooperatively combining the cache way (hardware) and set (OS) partitioning, SWAP is able to divide the shared cache into literally hundreds of regions, therefore providing sufficiently fine granularity for the upcoming manycore processor generation.

We implement SWAP as a user-space management thread on Cavium's ThunderX, a server-grade 48-core processor with ARM-v8 ISA [39]. To enable SWAP, we introduce small changes to the Linux page allocator, and leverage ThunderX's native architectural support for way partitioning.

Our results show that SWAP improves system throughput (weighted speedup) by 13.9%, 14.1%, 12.5% and 12.5% on average for 16-, 24-, 32- and 48- application bundles with respect to no cache management. This is twice as much speedup as what we can obtain by using only ThunderX's way partitioning mechanism.

To our knowledge, SWAP is the first proposal of a fine-grained cache partitioning technique that requires no more hardware than what's already present in commercial server-grade CMPs.

The paper is organized as follows: Section 2 provides background and comments on related work. Section 3 describes SWAP's mechanism and design challenges. Section 4 discusses the hardware and software implementation. Section 5 explains our evaluation framework, and Section 6 evaluates our proposal.

II. Background and Related Work

A. Way Partitioning

In the single core environment, Albonesi first proposes turning off unneeded cache ways to reduce cache energy [1]. Yang et al. improve such technique by dynamically adjust the active cache capacity to accommodate the changing working set of an application [44], [45].

In the multicore context, Suh et al. [38] propose to distribute L2 cache ways to minimize the overall miss rate. Qureshi and Patt [30] improve their technique by predicting the marginal utility of additional cache ways.

Way partitioning is a desirable approach because: (1) each core can be assigned an independent slice of the cache space, thereby reducing cache interference among co-running applications; (2) adjusting allocations is relatively inexpensive and can be accomplished lazily (i.e., cache lines in ways no longer part of an application's partition can still be accessed in place, until they are evicted). As a result, chip manufacturers have begun to adopt such a technique into their server processors [15].

Despite all these advantages, a well-known limitation of way partitioning is that it cannot by itself support more than a handful of applications at a time [33]. This is because cache associativity cannot scale easily with number of cores, as physical constraints result in increased latency and energy consumption.

B. Page Coloring

Page coloring [40] has been extensively used in industry and research community to improve the performance of the memory hierarchy. Kessler and Hill were among the first to use page coloring to improve the utilization of hardware cache, by distributing the physical pages evenly to different cache sets [19]. Such technique was later adopted by commercial OS such as FreeBSD [12]. A number of follow-up works further improve the cache utilization, and reduce the overhead of page recoloring [35]. For multi-core chips, Lin et al. [22] use page coloring to partition the shared cache among the cores in a dual-core chip to improve system efficiency. There are also proposals to use page coloring to partition memory banks [23], [47], or even to manage cache and memory contention cooperatively [24].

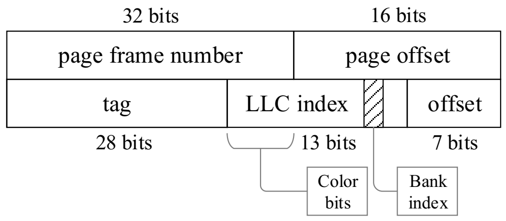


Fig. 1: Example of physical address mapping for page coloring, corresponding to Cavium's 48-core ThunderX architecture used in this study.

Instead of partitioning the cache "vertically" as in way partitioning, page coloring partitions the cache "horizontally" by sets. When an application requests a new page from the system, the OS will select a free page from its memory pool, and map the application's virtual address to the physical address of the page. In doing so, the OS may select a page frame whose page frame number (PFN) is of the appropriate "color"—the overlapping bits between the page frame number and last-level cache's set index (Figure 1). By constraining the color bits of the pages belonging to an application in this way, the OS may constrain an application's cache use to a subset of the cache sets.

Unfortunately, page coloring is hardly scalable, and it can incur significant overheads if recoloring is needed. Consider, for example, that a PFN's default size of 64KB allows for four color bits in Cavium's 48-core ThunderX (Figure 1). Sixteen colors is hardly sufficient to provide adequate isolation across 48 cores. One might consider reducing the page size to increase the number of colors, however this is typically counterproductive in the server market [16].

Even if a small page size were practical, page coloring still may not be able to provide fine granularity by itself: A well-known limitation of page coloring is that, by imposing page color restrictions on an application, only a portion of the system memory is accessible to this application [48]. This may result in an out-of-memory (OOM) error, even though the system may be awash with pages of other colors.

Finally, re-partitioning the cache space by page coloring is a costly process: If a page color is taken away from one application, all the associated page frames have to be migrated to page frames in the application's other colors, the appropriate TLB and cache entries flushed, etc.

C. Probabilistic Cache Partitioning

To address the scalability issue, Sanchez and Kozyrakis propose Vantage [34], a replacement-based partitioning mechanism that can probabilistically guarantee the size of partitions across applications. Wang and Chen [41] adopt a similar approach to maintaining a partition's cache size, by controlling the eviction priority of cache lines belonging to different cores. Although their simulation-based evaluations show promise, both proposals require a non-trivial amount of additional hardware support. For that same reason, their results cannot be validated by real implementations using commercial processors, where significant discrepancies could arise [22]. Finally, it is unclear whether these probabilistic approaches would be good enough for environments where strict isolation is highly desirable (e.g., to reduce exposure to timing channel attacks).

D. Cache Partitioning for Tile-based CMPs

In tiled-based architectures, each cache tile constitutes the primary container for the local core, and thus a natural partition exists. Lee et al. propose CloudCache [21], which explores allocating partitions potentially larger than tiles by "borrowing" cache ways from remote tiles. Beckmann and Sanchez propose Jigsaw [4], which improves upon CloudCache by favoring neighboring tiles, so as to minimize on-chip network latency.

However, tile-based architectures are still hard to come by, and tile-based caches present design challenges of their own. The fact is, most commercial CMPs with 20+ cores still implement a centralized (albeit banked) last-level cache organization (e.g., Intel's Xeon, IBM's Power and Blue Gene/Q, Cavium's ThunderX, etc.) Moreover, as in the case of probabilistic approaches, both techniques again introduce a non-trivial amount of hardware overhead, and they have yet to be supported in the commercial processors. This makes validation of simulated results very difficult.

III. Mechanism

SWAP combines both set and way partitioning, and in that way we can partition the shared last-level cache in a two-dimensional manner into many tens or even hundreds of regions, and then assign those regions to running applications. In Cavium's ThunderX 48-core processor, for example, the number of cache ways and possible page colors is 16 each. Therefore, ThunderX's shared L2 cache can be partitioned into 256 independent regions. Note that, theoretically, assignments may be chosen to overlap, but there is sufficient granularity to keep them disjoint, which is generally preferable for the reasons already stated.

A. Challenges

Although combining set and way partitioning to enable fine-grained cache partitions may be intuitive, in practice several important challenges needed to be addressed to make it practical. We discuss these next.

Partition placement. Correctly allocating a partition involves more than just picking the right size. On the one hand, partitioning by cache ways and page colors constrains the possible shapes and sizes of each cache partition. For example, in the ThunderX processor, it is infeasible to create a partition of 17 cache regions in the L2 cache with 16 ways and 16 colors. On the other hand, given a desired partition size, there may be multiple possible combinations of sets and ways to form a rectangle with that size. For example, 4×4, 2×8, and 1×16 allocations all offer the same capacity. Even if the partition size and shape of each application is feasible and known, placement of the partitions is a challenge in its own right.

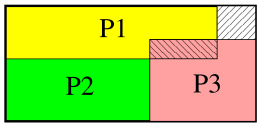


Fig. 2: An example of misaligned cache partitions that, on the one hand, leaves some cache space unassigned while, on the other hand, it forces some assignments to overlap.

Figure 2 shows an example of partitions that are not successfully placed. Note how there is some wasted cache space on the top right corner, and some overlap between partitions of P1 and P3, resulting in cache interference between the two. As part of our proposed solution, we describe later how we optimize the choice of partition placement.

Memory Pressure. As discussed in Section II, page coloring not only limits the number of cache sets an application can use, but also the amount of physical memory that it can access. The memory system could be awash with free physical frames of a particular color, and yet those would not be available to other applications that have been assigned a different color. Because SWAP employs page coloring, it is potentially subject to this problem. Although many colors potentially enable a fine-grain management of cache set allocations, it constrains each application to a small slice of the physical memory. Fortunately, this is not a major concern for SWAP, because SWAP adopts a coarse-grained page coloring approach, achieving fine granularity by combining it with way partitioning. In the ThunderX platform we study, for example, each page color covers 4GB of the 64GB available main memory, and we assign at least two page colors to each application (Section III-B). As a result, we never observed out-of-memory exceptions in any of our experiments.

Recoloring Overhead. Another major concern of page coloring is the potentially heavy cost associated with dynamic recoloring. When a color is taken away from an application, for example, all the pages with that color from that application have to be remapped across the remaining assigned colors. Page remap operations are cumbersome: they involve TLB and cache flushes, a page copy from its old memory location to the new one, and an update of the corresponding page table entry. Although efforts have been made to alleviate such overhead, for example by performing "lazy" page migration, recoloring overheads are generally non-negligible [22]. Therefore, SWAP needs to be carefully designed to avoid giving/taking away colors to/from applications whenever possible.

Increased Conflict Misses in Way Partitioning. One disadvantage of cache way partitioning is that it reduces the effective cache associativity of each partition, potentially increasing the number of conflict misses [34], [41]. Because SWAP inherits this disadvantage, we investigate the relationship between execution time and the number of cache ways in the context of our experimental setup (Section V), by statically sampling 30 different cache way+color configurations, with {1, 2, 4, 8, 12, 16} cache ways and {2, 4, 8, 12, 16} page colors. As a result, the effective cache capacity ranges from 128KB to 16MB. We find that, if the cache partition is formed by only one cache way, the number of conflict misses increases dramatically, and therefore the application's execution time suffers by up to 40% increase. On the other hand, for most applications, as long as their assigned partition has more than two cache ways, their performance is largely determined by the size of the assigned partition.

B. Algorithm

We propose a novel cache allocation mechanism to address these challenges. The mechanism starts by collecting the miss-ratio curve (MRC) of each application. The way the MRC is collected, whether using offline data or an online profiler, is orthogonal to the mechanism and has been addressed elsewhere [8]–[10], [30]. It then runs the *lookahead* algorithm proposed by Qureshi and Patt [30] to decide the optimal partition size of each core (in the unit of cache regions), so that the sum of partition size of each core is the total cache capacity. Note also that we guarantee 2 regions of cache space (128KB) for each core. More details will be explained in Section V. Note that the lookahead algorithm only determines the size of each partition given the total cache capacity, not how these are achieved in terms of ways vs. colors; this will be decided later by our placement algorithm, which we describe next.

1) Cache Partition Placement: An ideal partition should satisfy the following requirements: (1) partitions are aligned well with each other, without any wasted or overlapping cache regions, and (2) dynamic resizing should affect the fewest number of partitions during phase changes.

In SWAP, cache partitions are classified into multiple coarse-grain classes according to their size. Those in the same class are given the same number of colors. If the size of a partition changes within the range of its class, the number of colors it is given remains unchanged. The hope is that the partition may be able to keep its original page colors, to avoid any time-consuming recoloring. The general classification criterion for K colors ($K = 16$ in ThunderX) and S cache size (16MB in ThunderX), is as follows: (1) partitions of size larger than or equal to $S/4$ are afforded all K colors; (2) partitions whose size lies within $[S/8, S/4)$ are allowed $K/2$ colors; (3) partitions that fall within $[S/16, S/8)$ are assigned $K/4$ colors; and so forth, down to a minimum of two colors. In the case of ThunderX, this classification comprises four classes $C_i, i \in \{16, 8, 4, 2\}$, where i represents the number of colors assigned to applications in that class.

For placement (i.e., what specific colors each application receives), partitions with more colors are placed first and to the "left" (as represented by a rectangle of set rows by way columns) of partitions with fewer colors. Let us use an

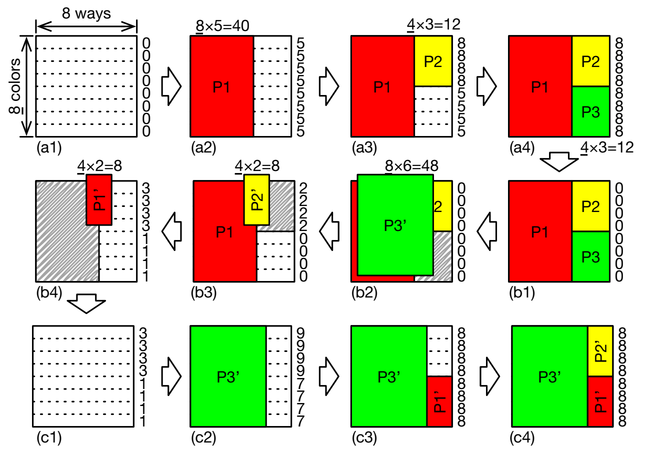


Fig. 3: A sample process of placing partitions based on their sizes and classes. The figure assumes eight colors and eight ways. The top row show an initial partition; center and bottom rows show the process of dynamically repartitioning based on changing application demands.

example (illustrated by the top row of Figure 3) to explain how this placement policy, combined with the classification criterion, can solve the alignment issue. In this example, P1 is an 8-color-class partition, and P2 and P3 are both 4-color-class partitions. Because P1 has more colors, it will always be placed to the left of P2 and P3. Thus, the right boundary of P1 and the left boundary of P2 and P3 are aligned.

Based on these two policies, the placement algorithm works as follows:

1. Each color maintains a "usage" counter (initialized to 0, as is shown in step a1 of Figure 3), which measures the number of cache ways of that color which have been already assigned.

2. The partitions are first classified into different classes according to the criteria mentioned above. Then the number of cache ways is trivially computed, by dividing the partition size by the number of colors, rounded down to an integer (not necessarily a power of 2).

3. We place cache partitions, in order from larger to smaller. For each partition, SWAP tries to find a set of consecutive colors with as little usage as possible. After the set of colors is determined, the partition will update the usage counter of its assigned colors, and the algorithm will move on to the next partition. Step a2 to a4 of Figure 3 shows such an example with 8 cache ways and 8 colors, where the sizes of the three partitions, P1, P2, and P3, are 40, 12, and 12, respectively. Because P1 is a C_8 partition, it is assigned all 8 colors. The usage counter of all 8 colors are updated to 5, since P1 receives five ways. SWAP then arbitrarily picks (since all colors show identical usage) the top four colors for P2 (since P2 is in class C_4), and it increases their usage counter by 3 each (the number of ways allocated to P2). Finally, when looking to place P3 in class C_4, SWAP picks the remaining four colors, again in this

case updating their counter to 3 each, since that is the number of ways allocated to P3 as well.

4. If a core is given the minimum cache space (2 regions), SWAP assigns one way and two colors to it, which may significantly hurt its performance due to conflict misses. As a result, in that case we may horizontally coalesce two or more minimum-sized partitions, using the same set of colors. Although coalescing may introduce some interference, we experimentally observe that it greatly reduces conflict misses (and that this kind of applications are often cache-insensitive anyway).

C. Reducing Recoloring Overhead

Our algorithm for recoloring strives to minimize color re-assignments. Specifically: (1) if a partition stays within its class, it should stick with its prior color assignment; (2) if a partition is downgraded to a class with fewer colors, its new colors should be a subset of its prior color set, so that only the "orphaned" pages need to be migrated; (3) if a partition is upgraded so that more colors are made available to it, it should attempt to add the set of colors that have the least "pressure" (smallest usage counter) at the time the partition is (re)placed. We continue to use Figure 3 (middle and bottom rows) to show this repartitioning process, where the sizes of P1, P2, and P3 and changed from 40, 12, 12 regions, to 8, 8, and 48 regions, respectively.

1. The partitions are pigeon-holed into different classes as before.

2. We reset the usage counters (step b1), and then estimate new usage for each color as follows: (1) If a partition is upgraded to a class with more colors (P3 in the example) , we do not increase the usage of any color (step b2). This is because the new partition, which will be handled earlier in the upcoming placement sequence, will explicitly seek to expand into the least used colors anyway. (2) If a partition stays in the same class (P2), we increase the usage of each color by the number of ways the partition will receive (step b3). This is to discourage other partitions that migrate into the same class from occupying these colors during placement. The goal is to allow applications that stay in the same class to keep their colors. (3) If the partition's class is downgraded (P1), the estimated new usage of the colors it currently maps to is increased by the number of cache ways the partition will receive, multiplied by the ratio of new to old number of colors for that partition. For example, P1 previously owned all 8 colors in Figure 3, but it is now downgraded to a 4-color class. The usage counter of colors 0-7 will be updated by the number of ways the new partition will receive, multiplied by 0.5. The rationale here is that the new partition will subset all former colors with equal probability, so on average each such color will see its usage affected equally and proportionally to the new allocation. (Recall that at this point we still do not know *which* colors will be picked.)

3. We start placing the cache partitions from larger to smaller, following the original algorithm, only that the expected usage is already initialized as explained above, and thus not computed from zero, but adjusted during actual placement. For example, in step c3 of Figure 3, we assign colors 4-7 to downgraded partition P1, because those colors show lower estimated usage (since colors 0-3 are "reserved" by P2). On the other hand, as a partition that remains in its same class, P2 will again pick its former colors (step c4). After placing each partition, usage for each color is adjusted to reflect the actual usage by that partition, by compensating with respect to the estimated usage previously calculated and accounted for, as is shown in step c3.

It is possible that a partition whose class is either unchanged or downgraded may find the expected usage of its previously assigned colors high enough that the partition may not be able to get the number of cache ways it needs. In that case, we allow the partition to move to a new set of colors that can accommodate its size; specifically, the partition will seek to move to a set with minimum calculated usage.

IV. Implementation

In this section we describe the existing hardware support that we leverage to implement SWAP, the software changes that we make to the operating system, and the interaction between them.

The ThunderX 48-core CMP is an ARM-based processor aimed at the server/datacenter market. It provides the ability to allocate the shared L2 cache by cache ways, up to 16 partitions. ThunderX provides a special register per core, which specifies the cache ways that a core can *insert* cache lines into. (Cores can still *access* lines in any cache way.) Once cache ways are assigned to cores (see Section III-B), SWAP configures the per-core registers so that the assignment may be enforced.

In order to further partition the cache by sets, we implement page coloring [22], [24], [47] in the Linux kernel that runs on the ThunderX system, by modifying its buddy memory allocator to fit our needs. We color user pages only; kernel pages are allocated using Linux's default mechanism.

In the buddy system, free physical pages are stored in multi-level free lists, where the kth-order free list contains pages which is composed of 2^k consecutive 64KB pages. We create multiple *bins* out of each list, with each bin caching pages of a specific color.

When a page fault occurs to a user application, the kernel first selects a page color in a round-robin fashion among all the allowable colors for that application. Then, it fetches a page of that color. When a bin is running out of pages, SWAP requests more free pages from the Linux buddy system and uses them to refill the bins.

A potential issue with page coloring is that some of modern processors adopt *hashed indexing*, where the index to the last-level cache is XORed with bits in the physical address [23]. Fortunately, because the physical address of a free page is readily available in the kernel, its color can be easily computed by hashing the appropriate bits.

TABLE I: CMP configuration.

ThunderX CN8800 [2], [6]	
Number of Cores	48
Frequency	2.0GHz
L1 ICache	78 kB, 128B cache line size
L1 DCache	32 kB, 128 cache line size
L2 Cache	16 MB, 16-way set associative, 128B cache line size
Memory Controller	64GB, 4 channels, DDR4 2133, aggressive bank reordering

SWAP works well with the large page sizes often found in server settings–in ThunderX's case, 64KB. It can also work well with smaller page sizes (e.g., standard 4KB pages), as long as the number of bits assigned for coloring is kept small, to skirt the issues of memory pressure and recoloring overhead described before. SWAP *as is* would not be able to leverage page coloring for very large "superpage" sizes supported in some architectures (e.g., 512MB for ThunderX), as the page offset would be very long, and therefore there would be no overlap between the page number and the cache index. Very large superpages are usually relegated to the uncommon case of servers with terabytes of physical memory [17], and produce undesirable side effects [11], [18], [20], [27], [29]. For example, database vendors often recommend users to turn off large superpage support [20], [27], because many database workloads tend to exhibit sparse rather than contiguous memory access patterns. Large superpages may also cause the system to run out of memory [11].

We follow a lazy approach to page migration for dynamic recoloring [22]: When a color is added to or taken away from an application, we eagerly walk through the application's page table and redistribute the application's pages across the colors assigned to it. For each page marked for migration to a different color, we reset the access flag (AF) in the page table entries (PTE) of the application's pages of that color, and set one other unused bit in each such PTE (we call it the Pending bit). Naturally, the corresponding TLB and data cache entries are also flushed. However, the application's marked pages are not immediately migrated. Rather, as pages for that application with AF=0 are accessed (which generates a page fault), if the Pending is set, the page will be migrated to its new color at that point (and the Pending bit will be reset). Then, the AF bit will be set, and the page fault handler will complete.

V. Experimental Setup

A. Hardware Platform

We evaluate SWAP on a Cavium ThunderX CN8800 rack server. The configuration of the processor is shown in Table I.

ThunderX supports hardware cache way partitioning, as is described in Section IV. We also develop a set of microbenchmarks similar to what Saavedra et al. [32] propose to verify the specifications related to the memory hierarchy (cache capacity, associativity, etc).

In addition, we check whether there is an overlap between the color bits and the memory channel and bank bits, as page coloring may restrict a core's accessibility to the memory channels/banks. To do this, we run the microbenchmarks proposed by Yun et al. [47] to detect the location of those bits, and we find that the memory channel and bank bits reside within the page offset, and therefore there is no overlap with the color bits.

TABLE II: Multiprogrammed workloads evaluated for simulation. Combining cache-insensitive (I), cache-sensitive (S), and thrashing (T) applications.

MP1	vpr - twolf - art - lbm	S^4
	vpr - ammp - bzip2- libquantum	S^2T^2
MP2	milc - soplex - lbm - art	T^4
	leslie3d - bwaves - GemsFDTD - bzip2	T^2S^2
MP3	vpr - twolf - milc - libquantum	S^4
	ammp - bzip2 - bwaves - soplex	T^4
MP4	mcf - milc - libquantum - leslie3d	T^4
	bwaves - GemsFDTD - twolf - swim	T^4S^2
MP5	mcf - soplex - libquantum - leslie3d	T^4
	bwaves - lbm - swim - art	T^2S^2
MP6	gamess - hmmer - milc - mcf	I^4
	tonto - h264ref - lbm - art	T^2S^2
MP7	twolf - art - leslie3d - bwaves	S^4
	bzip2 - mcf - GemsFDTD - libquantum	T^4
MP8	vpr - twolf - libquantum - milc	S^4
	ammp - art - mesa - sixtrack	T^2I^2
MP9	twolf - vpr - lbm - libquantum	S^4
	bzip2 - omnetpp - mesa - gobmk	T^2I^2
MP10	milc - soplex - h264ref - vpr	T^4
	libquantum - leslie3d - perlbench - mcf	S^2I^2

B. Software Platform

We prototype SWAP in the ThunderX platform running Ubuntu Trusty Tahr 14.04 with kernel version 3.18.0. SWAP runs as a user space management process, which includes (1) the algorithm described in Section III-B to decide the allowable cache region of each application; (2) the ability to write hardware registers to reconfigure cache way partition, and to interact with the underlying Linux kernel for page coloring (the implementation details are described in Section IV); and (3) a performance tracking thread which is triggered every 2 seconds to read hardware performance counters, such as the number of L2 cache misses.

C. Workload Construction

We use a mix of 22 applications from SPEC2000 [36] and SPEC2006 [37] to create multiprogrammed workloads for evaluation. Each application is compiled natively to an ARM executable, using gcc 5.1.0 with -Ofast optimization. We classify the 22 applications into *Cache-sensitive (S)*, *Cache-insensitive (I)*, and *Thrashing (T)* using offline profiling, and then create ten 8-application bundles that consist of a mix of applications from these three categories, as shown in Table II. When the number of active cores exceeds the number of applications in a bundle, the bundle is replicated across the chip. For example, 4 copies of MP1 would run in a 32-core configuration.

SWAP needs an estimate of the application's cache miss rate vs. capacity curve (MRC), which is used by the lookahead algorithm to produce the optimal size of each partition (described in Section III-B). In server-class environments, profile information can be obtained efficiently in a variety of ways, as addressed elsewhere [8]–[10]. Alternatively, it could be collected using additional hardware support (e.g., UMON [30]). In this paper, we use the applications' miss-per-kilo-cycle (MPKC) profile, by sampling 30 different cache way+color configurations, with $\{1, 2, 4, 8, 12, 16\}$ cache ways and $\{2, 4, 8, 12, 16\}$ page colors (the effective cache capacity ranges from 128KB to 16MB).

Besides SPEC, we also use a latency-critical workload, namely *memcached* from Cloudsuite [14], to study how SWAP guarantees QoS. Due to the lack of 10Gbit Ethernet support, we run the memcached server and clients on the same chip to avoid Ethernet becoming the bottleneck. Although packets are not physically transmitted via Ethernet, they still go through most of the OS networking layers, and therefore the cache behavior of the memcached server remains the same. In addition, in order to guarantee isolation between clients and server, we allocate 2 exclusive cache ways to all the client threads, which we find is good enough to issue requests in a timely manner. As recommended by Cloudsuite, we run one instance of the memcached server with 4 threads, and the QoS target is set such that 95% of the requests are serviced within 10ms [7]. The memcached client runs with 8 threads, and we configure the issue rate to 190K requests per second[1].

D. Performance Metrics

We use weighted speedup and L1 miss latency to evaluate our fine-grained cache partition, both of which can be obtained by SWAP's performance tracking thread described above. Weighted speedup measures the overall system throughput [13]. It is the arithmetic mean of the ratio between IPC_i^{shared} and IPC_i^{alone} for all applications i, where IPC_i^{shared} is the IPC obtained while running application i in a loaded system, and IPC_i^{alone} is the IPC when running unmolested.

We also use L1 miss latency to show the source of performance improvement. L1 miss latency directly correlates with the number of cycles that processor pipeline is stalled by long-latency memory operations, and it is computed as L2 access latency $+$ L2 miss rate $\times$ memory latency. An effective cache management technique should not only decrease the L2 miss rate of each application, but also reduce the overall memory contention, which further improves the L1 miss latency, and thus the IPC.

VI. Evaluation

We evaluate our SWAP proposal against a Baseline configuration, where the shared L2 cache is freely contended by all 48 cores. We also compare SWAP with a best-effort cache way partitioning (WAY) and page coloring technique (SET).

Our evaluation is done in four scenarios: First, we study the case of static partitioning, where cache repartitioning is not needed. Second, we study a scenario with real-time evolving workloads, with applications coming and going, and where the dynamic cache partitioning is involved to react to the changing cache demands. We also study the overhead of the dynamic SWAP in the ThunderX platform. Third, we study how SWAP guarantees the quality of service (QoS) of latency-critical workloads, and improves the throughput of background batch applications at the same time. Note that all of the above experiments are done on a real ThunderX rack server. Finally, we compare SWAP with recently proposed probabilistic cache partitioning in the simulator.

A. Static Partitioning

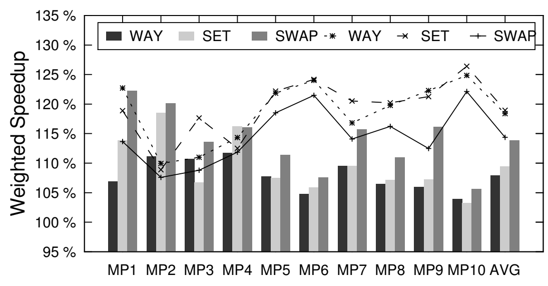
In the static partitioning experiments, we run SWAP on 16, 24, 32 and 48 cores of a ThunderX 48-core processor, with the applications bundles detailed in Section V-C. SWAP first reads the MPKC profile of each application, and then computes the size, shape, and placement of each core's cache partition in the shared L2, based on the algorithm discussed in Section III-B. When an application finishes before the whole bundle has finished, the same application is again instantiated on the same core. It naturally inherits the cache partition of the core, and therefore no repartition is needed. When all the applications have finished at least once, we kill all the processes and conclude the experiment. The purpose of this experiment is to measure the partitioning quality of SWAP.

We first compare SWAP with Baseline (no cache partitioning involved), and the results are shown in Figure 4. SWAP consistently outperforms Baseline for all the bundles in all configurations, and the improvement does not decrease with more active cores, showing a good scalability in large-scale CMPs. On average, SWAP improves system throughput over Baseline by 13.9%, 14.1%, 12.5%, and 12.5% for 16-, 24-, 32-, and 48-core configurations, respectively. We also find that, on average, SWAP reduces the chip's overall L1 miss latency by 31.3%, 30.1%, 25.7%, and 17.6%, respectively.

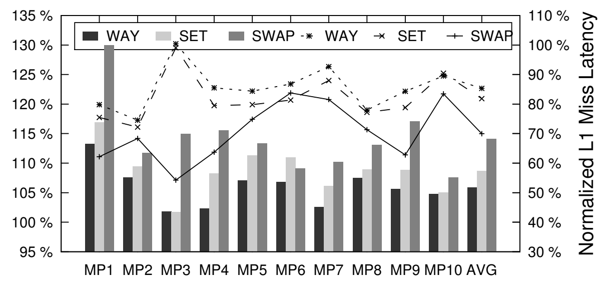
We also compare SWAP with utility-based way partitioning (WAY) [30] and page coloring technique (SET) [22]. Because there are only 16 cache ways or page colors in ThunderX, it is impossible to give a separate cache partition to each application in either mechanism, if the number of active cores is larger than 16.

We design a variation of way partitioning that allows for judicious sharing of cache ways for larger configurations as follows: We begin by reserving a small number of cache ways as the "dump area." Then, we run Qureshi and Patt's lookahead algorithm [30] to allocate the remaining cache ways. The lookahead algorithm iteratively finds the application that has the highest marginal utility on cache capacity, and assigns the cache ways to it. We run such algorithm until all the cache ways are allocated (except for the "dump" area). Then, all the remaining applications are assigned the "dump" area. In addition, as discussed in Section III-A, a partition with one cache way usually introduces an excessive number of conflict misses, hurting the application's performance. As a result, we adopt an approach similar to what Liu et al. propose [24], which coalesces the neighboring partitions if one of the them has only 1 cache way. Such coalescing rule greatly helps

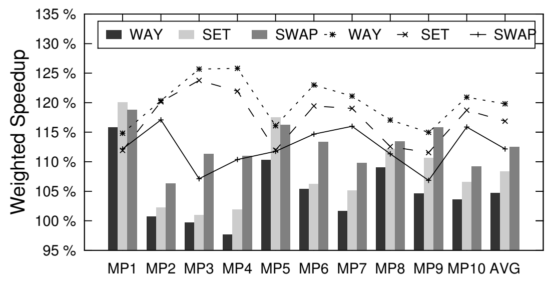
[1]We find that 190K is the maximum issue rate for the memcached server to meet its QoS target even if it is given the entire cache.



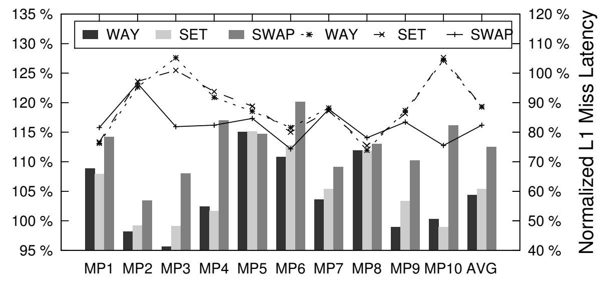
(a) 16 cores



(b) 24 cores



(c) 32 cores



(d) 48 cores

Fig. 4: Comparison of system throughput (weighted speedup) and L1 miss latency for Baseline, WAY, SET and SWAP. Both weighted speedup and L1 miss latency are normalized to Baseline. The bars show the weighted speedup, while the lines show the L1 miss latency normalized to baseline.

reduce conflict misses, and we find it significantly improves the performance of WAY.

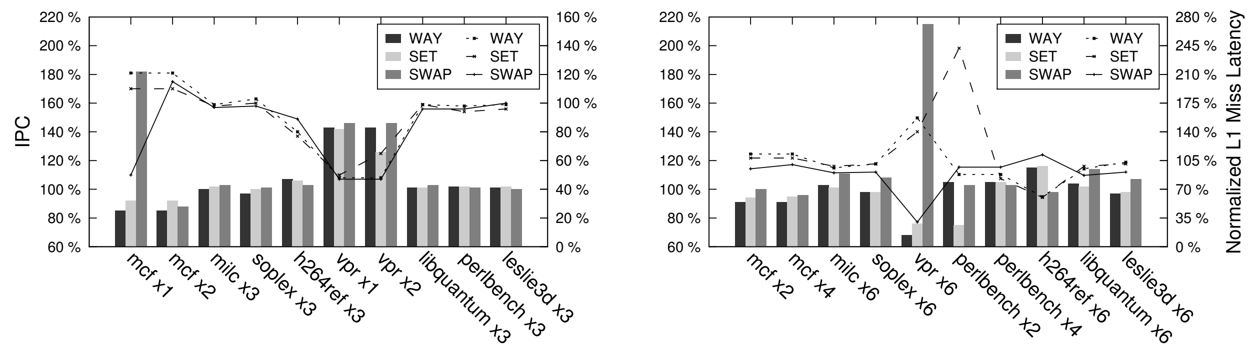
We also study the effect of varying the size of the "dump area" from 2 to 6MB. We find that in general, the "dump area" should be as small as possible. For 16, 24, and 32 cores, reserving 2 ways performs the best. However, for 48 cores, reserving 4 ways produces the most speedup because there are more than 30 applications in such "dump area."

Our coarse-grained page coloring scheme (SET) works similarly to WAY. However, because constraining the page colors also limits the amount of physical memory accessible by the applications, more colors should be reserved to avoid an out-of-memory error (OOM). Our study shows that reserving 2, 4, 4, and 6 colors can prevent OOM and produce the most speedup for 16-, 24-, 32-, and 48-core configurations respectively.

Figure 4 shows the weighted speedup of WAY, SET, and SWAP normalized to Baseline. Although WAY and SET perform well at small core counts, their partitioning quality degrades as the number of active cores increases. We look closely at a representative bundle MP10; Figure 5 shows the normalized IPC and L1 miss latency of each application in the bundle in 24- and 48-core configurations. In the 24-core configuration, both SET and WAY can provide a 2MB partition for each instance of the cache sensitive application vpr. SWAP, on the other hand, can provide a tighter 1.5MB partition to each instance of vpr, and the resulting savings are given to another sensitive application, mcf (whose partition is in the dump area under SET and WAY). Although this improves the overall system throughput by only 2% in the 24-core configuration, the effect is amplified at higher core counts. Moreover, by reducing the overall L2 miss rate, SWAP greatly alleviates memory contention in the 48-core configuration, and thus helps even the non-sensitive applications. As a result, SWAP leads SET and WAY by 16%. Overall, SWAP outperforms WAY and SET by 4.36%, 5.44%, 4.3%, and 7.14%, respectively, for 16-, 24-, 36-, and 48-core configuration.

B. Dynamic SWAP with Changing Workloads

It is not necessary to invoke cache repartition in the static experiments so far, because the application running on each core is fixed for each bundle. In this section, we evaluate SWAP in a dynamic scenario where workloads that come and go. Again, we keep the number of active cores to be fixed (16, 32, and 48 for this experiment). Instead of running a fixed bundle, we generate a long sequence of SPEC applications, and we inject the applications from the top of the sequence to the system until the number of active cores reaches the desired number. When an application finishes, we fetch the next application from the sequence, and schedule it to the currently idle core. This is similar to the scenario in clusters or data centers, where a sequence of applications is waiting in the task queue



(a) 24 cores

(b) 48 cores

Fig. 5: The breakdown of a sample bundle MP10 running on 24 and 48 cores in ThunderX. The bars show the IPC (normalized to IPC_{alone}), and the lines show the normalized L1 miss latency of each application.

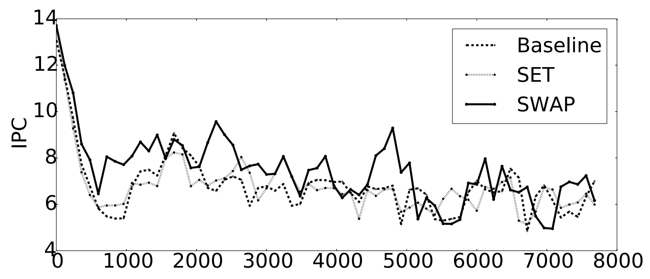


Fig. 6: Real time throughput of Baseline, SET and SWAP of Sequence 2 in the 48-core case over time (seconds).

for available cores. Because the new application may show different cache characteristics, dynamic cache repartitioning is desirable. For example, assume that an application with a large cache partition completes, and that a cache-insensitive one is introduced into the system. The unwanted cache capacity of the new application will be re-distributed to the other cores in the system, which triggers a system-wide repartition.

For our 16-, 32-, and 48-core configurations, we construct a sequence of 32, 64, and 96 applications, respectively, which contains a mix of applications in different categories (I, S, T). All the applications in the sequence have to finish at least once, and when all of them finish, we terminate the experiment and report the system throughput of the entire sequence. When the fetch reaches the end of the sequence, it will start over from the head of the sequence, and therefore no core will be idle.

We construct two sequences, both of which include 16 distinct SPEC benchmarks (out of 22 that we use in this paper). Table III shows the SWAP's improvement over Baseline and WAY in terms of weighted speedup. SWAP improves the weighted speedup by 8% and 17% for the two sequences in the 16-core cases, and the improvements increase to 11% and 20% for the sequences in the 32-core cases. Although WAY does fairly well in the 16-core sequences, its partition quality drops significantly beyond that. Besides the scalability issue of WAY

TABLE III: Comparison of system throughput (weighted speedup normalized to Baseline) for SET, WAY, and SWAP in the dynamic experiment.

Cores	Seq	WAY	SET	SWAP	Avg. Inj interval
16	1	1.04x	1.02x	1.08x	46s
	2	1.11x	1.04x	1.17x	41s
32	1	0.97x	1.04x	1.11x	31s
	2	1.04x	1.02x	1.20x	25s
48	1	0.92x	0.99x	1.11x	34s
	2	1.00x	1.03x	1.15x	25s

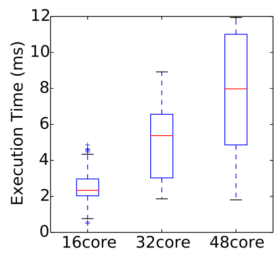
and SET discussed in the static experiment in Section VI-A, another important reason for the poor performance is that the application's injection rate is much higher (shown in Table III) with a higher core count. An application may be be frequently moved in and out from the "dump" area, which significantly hurts performance. Figure 6 shows the real-time throughput (sum of IPC) of SWAP vs. Baseline and SET. It is clear that SWAP outperforms Baseline and SET most of the time, as it "runs ahead" of Baseline and SET.

C. SWAP Overhead

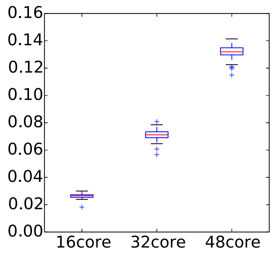
This section studies the overhead of our SWAP approach. The overhead comes from two sources: (1) the execution time of the SWAP algorithm, which decides the allowable cache region of each core; and (2) the overhead of page recoloring, which involves migrating pages of an application from its old colors to the new ones.

1) Algorithm Overhead: As is described in Section III-B, SWAP first runs the *lookahead* algorithm [30] to decide the optimal partition size of each core, followed by our proposed placement technique to decide the actual cache region. The complexity of the *lookahead* algorithm is $O(N^2)$, where N is the number of active cores in the chip. Our placement technique, which involves sorting all partitions by their size, has a complexity of $O(Nlog(N))$.

Figure 7 shows the distribution of the execution time of the SWAP mechanism in 16-, 32-, and 48-core configurations. As is shown in Figure 7a, on average, the overall SWAP algorithm



(a) SWAP



(b) Placement

Fig. 7: Execution time distribution of the overall SWAP, and the placement algorithm (i.e., what specific colors each application gets) in 16-, 32-, and 48-core CMP.

consumes 2, 6, and 8 ms for 16-, 32-, and 48-core CMP (12ms in the worst case), which is negligible compared with the 25s repartition interval. Figure 7b shows the execution time of our placement technique (i.e., what specific colors each application gets) across different configurations. Although it increases linearly, it takes less than 0.15 ms even for the 48-core configuration.

2) Recoloring Overhead: Our SWAP algorithm tries to avoid recoloring by taking the previous color assignment into consideration. However, recoloring is sometimes unavoidable to produce high quality cache partitions, and therefore we study the overhead of recoloring by micro-benchmarking. In the micro-benchmark, we actively recolor 50% of the pages for each SPEC application every 20s, and record the system time of that application. We consider the system time to be the aggregated overhead of page recoloring. [2] Table IV shows the overhead of some sample applications. The overhead per recoloring heavily depends on the number of pages being migrated. For the applications with a large memory footprint (e.g., bwaves migrates 70K pages), the overhead is about 200ms. For the applications that migrate a few thousand pages, the overhead is negligible. In any case, the overhead is small compared with the 25s application repartition interval in our setup.

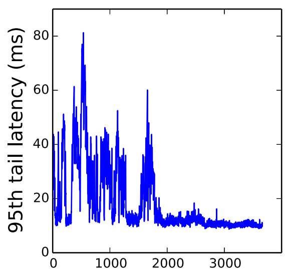
TABLE IV: Recoloring Overhead

app	total # page recolored	overhead per repartition (ms)
bwaves	71400	213.00
leslie3d	8000	28.00
bzip2	4900	11.00
gobmk	1350	8.00
gromacs	1100	4.00

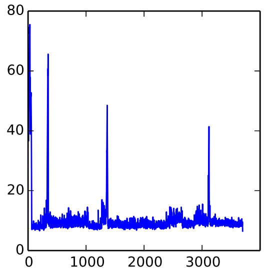
D. Providing QoS Guarantees

A number of studies have found that the utilization of most datacenter servers are low, and a primary reason is that the load of popular latency-critical (LC) workloads varies significantly due to diurnal patterns and unpredictable spikes in user accesses [3]. A promising way to improve server utilization is to launch batch workloads on the background to exploit the unused hardware resources [25]. A key to this approach is that the QoS of the LC workloads should not be affected by the batch workloads. In this section, we propose to use SWAP to maximize the background batch workloads, with a prerequisite of guaranteeing the QoS of a popular LC workload memcached.

[2]This is a conservative estimation, because we account all the system time to be the overhead of recoloring.



(a) Shared cache



(b) Exclusive cache

Fig. 8: Real time 95th tail latency of memcached co-running with 16-app bundle MP1 over wall clock time (second), with QoS as 10ms.

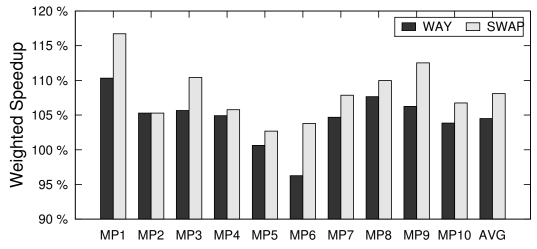


Fig. 9: Comparison of system throughput (weighted speedup) of the background 16-app bundle for WAY and SWAP, when the QoS of memcached is satisfied. Weighted speedup is normalized with Baseline.

The memcached setup is described in Section V-C. We use the multi-programmed 16-app SPEC bundles detailed in Table II as the batch workloads. In order to guarantee QoS of memcached, we allocate an independent cache partition to the memcached server to avoid interference. In addition, the capacity of such partition has to be dynamically adjusted to satisfy the QoS. We adopt a feedback-based mechanism similar to the one proposed by Lo et al. [25], which reads the tail latency every 30s. We start with two cache ways for the memcached server, and when the QoS is not met, we increase the size of its partition by one cache way. When QoS is met for a period of time (10 minutes in our setup), we decrease the partition size to explore whether the QoS can still be met. Figure 8 shows the tail latency of memcached over time when the server either shares cache with a sample 16-app SPEC bundle MP1, or owns its exclusive cache partition whose size is dynamically adjusted. We find that QoS (95th tail latency at 10ms) is frequently violated in the case of shared cache, but is satisfied most of the time in the exclusive cache case. A few spikes exist in Figure 8b because: (1) the background applications exert higher memory pressure due to phase change, which increases the penalty of L2 misses that is no longer tolerable by memcached; (2) the partition size of memcached is reduced for exploration (discussed above). In either case, our feedback-based mechanism reacts fast enough to reduce the tail latency to normal.

We use SWAP and WAY to partition the remaining cache capacity among the background SPEC applications, and compare them with a Baseline where all the SPEC applications share the cache capacity left by memcached.[3] Note that the partition of memcached server can only be adjusted by cache ways, because the overhead of recoloring its pages is too much to guarantee QoS. As a result, we exclude SET in this study. Figure 9 shows the system throughput of ten 16-app bundles managed by WAY and SWAP. Although memcached occupies a non-trivial amount of cache space, SWAP is still able to provide enough granularity to partition the cache space, resulting in 8.10% improvement in system throughput on average over Baseline. This almost doubles the improvement of WAY, which suffers from the limited granularity.

E. SWAP vs. Probabilistic Cache Partition

Probabilistic cache partition mechanisms [34], [41] have been proposed as a scalable cache management technique for large CMPs. However, to the best of our knowledge, all those proposals require non-trivial hardware changes that are currently unavailable on real processors. Therefore, in order to compare against probabilistic cache partition mechanisms, we implement SWAP in architectural simulator SESC [31]. However, we run into a dilemma: on one hand, simulation is multi-order of magnitude slower than real machine execution, and we can only simulate 100M instructions due to the time constraints. This is equivalent to less than 1 second of actual execution, which is almost negligible compared with hours of running in our real machine experiment; on the other hand, the overhead of SWAP is at the order of milliseconds, and we have to simulate long enough to amortize this overhead. As a result, in SESC, we ignore the two sources of SWAP overhead described in Section VI-C, and focus on whether SWAP is able to provide the same quality of management as those probabilistic cache partition mechanism. Other overheads, such as cache and TLB flush due to page migration, are faithfully modelled. Note that the results of our real machine studies include all the SWAP overheads.

We compare SWAP with traditional Unmanaged LRU policy; Futility Scaling [41], which is a recently proposed probabilistic cache partition mechanism that maintains fine-grained partition using a feedback control mechanism; and utility-based way partitioning (UCP) [30] with a highly-associative cache. The architectural configuration is the same as ThunderX processor described in Section V-A, with 32 active cores. The shared L2 cache is 16MB with 16 ways, unless specified otherwise. The application bundles are the same as the previous real machine study.

[3]SWAP recolors pages that belong only to SPEC applications.

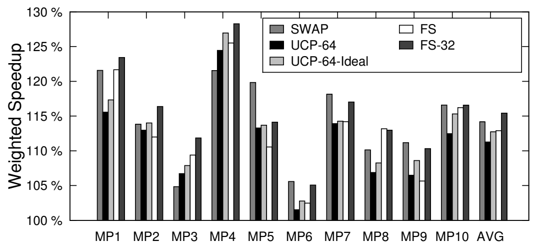


Fig. 10: Simulated results of a simplified SWAP model (Section VI-E) vs. Futility Scaling (FS) and Utility-based Cache (way) Partitioning (UCP) with a highly associative cache.

Figure 10 shows the system throughput (weighted speedup), normalized to Unmanaged LRU policy. We first compare with UCP, which partitions the cache by ways. To give an independent partition to each core, we evaluate UCP with 64 cache ways, for which Cacti [28] reports a 25% increase in access latency compared to 16 cache ways in ThunderX. UCP-ideal assumes the same access latency. Figure 10 shows that SWAP outperforms UCP in 8 out of 10 bundles, which shows that SWAP with 16 cache ways provides a finer granularity than UCP with 64 ways. The reason why UCP slightly outperforms SWAP on bundle MP3 and MP4 is that SWAP constrains the shape of each partition, thus not all partition sizes are allowed (e.g., the partition size is a multiple of its number of colors).

We then compare SWAP with Futility Scaling (FS) [41]. FS maintains a "futility" index of each cache line, and evicts cache lines with the maximum futility among the replacement candidates in the same set. Theoretically, FS is able to maintain the partition size at the granularity of lines. However, we find SWAP outperforms FS for 7 out of 10 bundles. This is because with 16 cache ways, the number of replacement candidate (16) is not large enough to include all the lines with large futility indices. As a result, we evaluate FS with 32 cache ways (FS-32), and we find that SWAP achieves comparable performance improvement. In addition to requiring fewer cache ways, SWAP does not require any extra hardware and is readily available in commercial processors, without giving up any performance improvement.

VII. Conclusion

We have proposed SWAP, a fine-grained cache management technique that seamlessly combines set and way partitioning with minimum hardware support. SWAP can successfully provide hundreds of fine-grained cache partitions to achieve effective cache partitioning in the manycore era. We have prototyped SWAP on a real 48-core Cavium ThunderX-based machine running Linux, and shown average speedups over no cache partitioning that are twice as large as those attained with way partitioning alone.

Acknowledgments

The authors would like to thank Bryan Chin and Srilatha Manne for their valuable insight, and the anonymous reviewers for their feedback. This work was supported in part by AFOSR Award FA9550-15-1-0311; a research contract with, and an equipment gift from Cavium; and a gift from Qualcomm. Jeff Setter contributed to this work as an undergraduate student at Cornell, and he was supported in part by Cornell's Engineering Learning Initiatives program.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Intl. Symp. on Microarchitecture (MICRO)*, 1999.

[2] AnandTech. ARM Challenging Intel in the Server Market: An Overview. http://www.anandtech.com/show/8776/arm-challinging-intel-in-the-server-market-an-overview/4, 2014.

[3] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[4] N. Beckmann and D. Sanchez. Jigsaw: scalable software-defined caches. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[5] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[6] Cavium Inc. ThunderX Family of Workload Optimized Processors. http://www.cavium.com/pdfFiles/ThunderX_PB_p12_Rev1.pdf, 2013.

[7] Cloudsuite. Cloudsuite. http://cloudsuite.ch/datacaching/.

[8] C. Delimitrou and C. Kozyrakis. The Netflix challenge: Datacenter edition. *Computer Architecture Letters (CAL)*, 2013.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[10] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[11] DigitalOcean. Transparent huge pages and alternative memory allocators: A cautionary tale. https://www.digitalocean.com/company/blog/transparent-huge-pages-and-alternative-memory-allocators/.

[12] M. Dillon. Page coloring optimizations. http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html.

[13] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE MICRO*, 28(3):42–53, 2008.

[14] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[15] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: from concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2016.

[16] IBM Inc. 64KB pages on Linux for Power systems. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Welcome+to+High+Performance+Computing+(HPC)+Central/page/64KB+pages+on+Linux+for+Power+systems, 2012.

[17] R. H. Inc. Huge pages and transparent huge pages. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html.

[18] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. Performance analysis of the memory management unit under scale-out workloads. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2014.

[19] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 1992.

[20] K. Kirkconnell. Often overlooked linux os tweaks. http://blog.couchbase.com/often-overlooked-linux-os-tweaks.

[21] H. Lee, S. Cho, and B. R. Childers. Cloudcache: Expanding and shrinking private caches. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2011.

[22] 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 *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2008.

[23] 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 *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[24] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[25] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Intl. Symp. on Computer Architecture (ISCA)*, 2015.

[26] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (PriSM). In *Intl. Symp. on Computer Architecture (ISCA)*, 2012.

[27] MongoDB. Disable transparent huge pages (thp). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[28] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, 2009.

[29] B. Pham, J. Veselỳ, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Intl. Symp. on Microarchitecture (MICRO)*, 2015.

[30] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Intl. Symp. on Microarchitecture (MICRO)*, 2006.

[31] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[32] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Micro benchmark analysis of the KSR1. In *ACM/IEEE Conf. on Supercomputing*, 1993.

[33] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Intl. Symp. on Microarchitecture (MICRO)*, 2010.

[34] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Intl. Symp. on Computer Architecture (ISCA)*, 2011.

[35] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Intl. Conf. on Supercomputing (ICS)*, 1999.

[36] Standard Performance Evaluation Corporation. SPEC CPU2000. http://www.spec.org/cpu2000/, 2000.

[37] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/, 2006.

[38] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[39] The OVH group labs. ARM Cloud. https://www.runabove.com/armcloud.xml, 2016.

[40] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Intl. Symp. on Microarchitecture (MICRO)*, 2001.

[41] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *Intl. Symp. on Microarchitecture (MICRO)*, 2014.

[42] X. Wang and J. F. Martínez. Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[43] X. Wang and J. F. Martínez. ReBudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[44] S. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2001.

[45] S.-H. Yang, M. D. Powell, B. Falsafi, and T. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2002.

[46] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: a dynamic cache partitioning system using page coloring. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[47] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[48] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European conference on Computer systems (EuroSys)*, 2009.