A Software Cache Partitioning System for Hash-Based Caches

ALBERTO SCOLARI, DAVIDE BASILIO BARTOLINI, and MARCO DOMENICO SANTAMBROGIO, Politecnico di Milano

Contention on the shared Last-Level Cache (LLC) can have a fundamental negative impact on the performance of applications executed on modern multicores. An interesting software approach to address LLC contention issues is based on *page coloring*, which is a software technique that attempts to achieve performance isolation by partitioning a shared cache through careful memory management. The key assumption of traditional page coloring is that the cache is physically addressed. However, recent multicore architectures (e.g., Intel Sandy Bridge and later) switched from a physical addressing scheme to a more complex scheme that involves a hash function. Traditional page coloring is ineffective on these recent architectures.

In this article, we extend page coloring to work on these recent architectures by proposing a mechanism able to handle their hash-based LLC addressing scheme. Just as for traditional page coloring, the goal of this new mechanism is to deliver performance isolation by avoiding contention on the LLC, thus enabling predictable performance. We implement this mechanism in the Linux kernel, and evaluate it using several benchmarks from the SPEC CPU2006 and PARSEC 3.0 suites. Our results show that our solution is able to deliver performance isolation to concurrently running applications by enforcing partitioning of a Sandy Bridge LLC, which traditional page coloring techniques are not able to handle.

Additional Key Words and Phrases: Hash-based cache, last-level cache, Linux, operating system, page coloring

1. INTRODUCTION

The number of cores within multicore architectures has constantly been increasing recently, thanks to the advances of lithographic technology. Therefore, co-location of multiple tasks on a single multicore is common on almost every modern platform, from smartphones to multi-socket servers. However, co-located applications contend for multicore shared resources, such as on-chip interconnection bandwidth, the mem-ory controller, I/O ports and, in particular, the Last-Level Cache (LLC) [Fedorova et al. 2010]. This last component is particularly important to ensure the performance of applications, which contention may severely degrade. In particular, co-located applications may conflict for LLC resources, such as ports and sets. If multiple cores access the same port simultaneously, the LLC can serve only one request at a time and must defer

Authors' addresses: A. Scolari and M. D. Santambrogio, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Via Ponzio, 34/5, 20133 Milan, Italy; emails: {alberto.scolari, marco.santambrogio}@polimi.it; D. B. Bartolini (current address), Oracle Software (Schweiz) GmbH, Täfernstrasse 4, 5405 Baden-Dättwil; email: davide.bartolini@oracle.com.

This is the accepted version of Scolari A., Basilio Bartolini D., Santambrogio M.D., A software cache parti-tioning system for hash-based caches. ©ACM Trans. Archit. Code Optim., 2016, 13:4.
The final publication is available at: http://dx.doi.org/10.1145/3018113

the requests from other cores, thus blocking their execution. Additionally, if multiple cores access the same set, they could cause the eviction of lines of other cores. This phenomenon happens because the different access patterns of different applications mix within the same set, and the Least Recently Used (LRU) policy is unable to detect and deal with different access patterns simultaneously. The little predictability of contention, in turn, makes it difficult to predict the performance of co-scheduled applications.

The clusters powering modern cloud computing are particularly subject to these issues, as management policy generally tries to co-locate workloads on the same multicore in order to pack the workload on a smaller number of servers. Cloud applications served by these clusters are often bound to provide Quality of Service (QoS) guarantees to customers, with penalties for the service provider in the case of QoS violation. In these environments, contention is a particularly important issue, and performance unpredictability has consequences at the business level.

Recently, manufacturers of multicores tackled some of these issues with architectural changes. For example, Intel tried to solve contention at the port level by deep changes to the LLC structure. With the Sandy Bridge family, Intel split the LLC into multiple parts, called *slices*, each with dedicated resources, that cores can access via a dedicated ring interconnection network [Lempel 2011]. To prevent multiple cores from accessing the same slice simultaneously and to avoid bottleneck effects, Sandy Bridge spreads accesses by means of a hash function computed on the physical address of the data to be retrieved; instead, the access within a slice leverages the physical address with the usual scheme.

This deep change, however, does not solve contention within sets. Cores can still access any slice uniformly, and co-located applications are still likely to experience strong contention when accessing sets within a slice. Intel, with the new Haswell platform, provides a solution based on way partitioning [Intel Corp. 2015], which comes at the cost of limiting the associativity available to each application. To overcome this limitation, several approaches are possible to decrease contention within sets, as discussed in Section 3. Among them, some propose software solutions based on *page coloring*, a technique that leverages the physical memory to control the mapping of data into the LLC: by controlling physical addresses of data, page coloring can partition the LLC among applications and increase the predictability of performance. However, the changes to the addressing scheme that Intel introduced affect the applicability of classic page coloring–based techniques. Resultant limitations are discussed in Section 4.

1.1. Outline and Contributions

To provide useful background for the rest of the article, Section 2 introduces the technique of page coloring and shows the *Buddy algorithm*, which is at the base of Linux memory management and is the starting point of the proposed design. With this information, Section 3 reviews related work on performance isolation within the LLC, showing the major hardware and software solutions. Section 4 shows how hash-based LLC addressing impacts page coloring, and outlines the design of the proposed solution. In particular, it explains how to reconstruct the multicore hash function and how to use this information to modify the physical memory management subsystem in order to implement page coloring on hash-based caches as well. Section 5 validates the proposed solution on a real hash-addressed LLC, showing how it can improve performance control and predictability, and highlighting actual limitations of our approach. Section 6 contains our conclusions, achievements of this work, and possible future research.

This article brings three main contributions.

(1) We provide a methodology to reverse engineer the hash function of Sandy Bridge CPUs. The proposed methodology leverages hardware performance counters

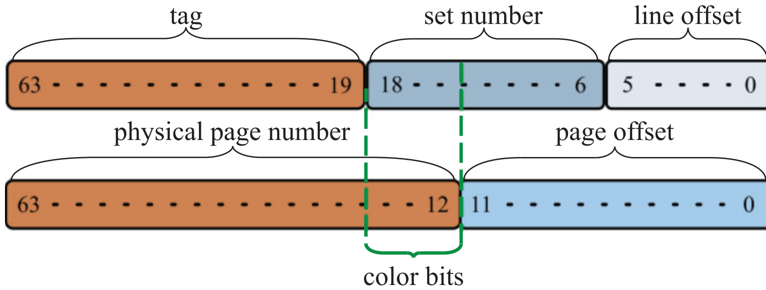

Fig. 1. Bit fields of a physical memory address.

available in modern architectures and is based on assumptions that previous work demonstrated to be consistent across multiple families. Unlike previous methods, our methodology is robust to noise in input data and identifies the exact hash function. Moreover, although we experimented with only one CPU model, we believe it is general enough to be applicable to recent models by Intel.

(2) Our solution uses the information on the hash to generalize page coloring to hash-based caches, allowing the system administrator to choose the size and the position of the LLC partition even in the presence of hash-based addressing.

(3) We generalize the notion of "page color" within the Linux memory allocator, thus achieving scalability and efficiency with a more general design than previous works.

We implemented our page coloring scheme in the Linux kernel and validated it on real hardware with workloads from the SPEC CPU2006 and PARSEC 3.0 benchmark suites.

2. BACKGROUND

This section introduces the fundamental concepts that this work is based on. Section 2.1 introduces *Page coloring*, the state-of-the-art technique to partition CPU caches, which requires software-only modifications. These modifications mainly affect the physical memory allocator of the operating system (OS), which is the Buddy allocator in Linux and is introduced in Section 2.2.

2.1. Page Coloring

Well-known in the literature, page coloring [Bray et al. 1990] is the mechanism at the base of techniques for software cache partitioning. It is based on the way modern shared caches map data to cache lines. These caches use the physical address to map data; the allocation position can thus be controlled via the physical memory allocator of the OS. Some bits used to select the cache set are typically in common with the address of the physical page, and can be controlled via the OS to reserve cache space for a given application.

For example, Figure 1 shows the parameters of a real CPU, an Intel Xeon E5 1410, in which the LLC is the third layer of cache. There, bits 12 to 18 are in common between the set number and the physical page number, and are called *color bits*, while a configuration of them is called *page color*. In the example, 7 color bits are present, so that the LLC can be split in at most 128 partitions. Nevertheless, Figure 3 shows that bits 12 to 14 are used both as color bits and as set bits for L2 cache addressing. Using these bits for LLC partitioning would also partition the per-core L2 caches of accessing cores, which is undesirable. Therefore, only bits 15 to 18 are finally available for partitioning, finally allowing 16 partitions.

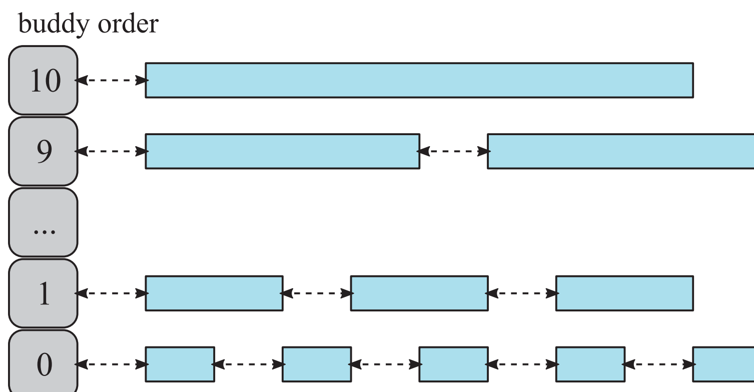

Fig. 2. Data structure of the Buddy algorithm, with one list of buddies per order.

To enforce the partitioning, the OS allocates pages on a per-application basis. Each application is assigned a suitable number of colors; the OS uses those colors only for the application memory. Therefore, other co-located applications cannot interfere with the data accesses of the target application, which is able to exploit the cache space reserved with full associativity.

The main disadvantage of page coloring comes when repartitioning (called "recoloring") occurs: in this case, memory pages must be copied to new locations of different colors, the consequence being a high overhead for data copy (in the order of 1us per page). Thus, page coloring has mainly been investigated as a static "technique," and the works in the literature try to limit recoloring overhead by copying pages only when really needed.

2.2. Buddy Algorithm

The Buddy allocator [Knowlton 1965] divides the physical memory into *buddies*, which are contiguous memory areas of different sizes. A parameter called *order* characterizes each buddy, whose size is 2^{order} the size of a page. Buddies are aligned to a memory address that is a multiple of their size, thus being aligned to *order* bits beyond the number of page bits (typically 12). Therefore, a buddy of order 0 is aligned to a 12b boundary, a buddy of order 1 is aligned to a 13b boundary, and so on. This alignment constraint allows identifying each buddy through the memory address of its first byte. Each buddy is strictly coupled to either the previous or the following buddy of the same size, depending on the least significant nonaligned bit. To find the coupled buddy from a given one, it is sufficient to invert this bit.

Figure 2 shows the data structure that the algorithm uses to achieve efficiency, storing buddies of different orders in different lists. Therefore, when subsystems of the OS request a memory area of a certain size, the allocator rounds it by excess to the closest power-of-two number of pages and returns the buddy of that size. In the case in which no suitable buddy is present, the buddy algorithm *splits* a buddy of higher order into two parts, stores the second half to the list of free buddies of lower order, and returns the first half. Conversely, the algorithm maintains scalability and efficiency by *merging* two contiguous free buddies, taking advantage of the "companion" of buddies. On every buddy being freed, the buddy algorithm checks whether the coupled buddy is also free and groups them in a single, higher-order buddy that is inserted in the proper list, eventually iterating the merging procedure until the maximum order is reached.

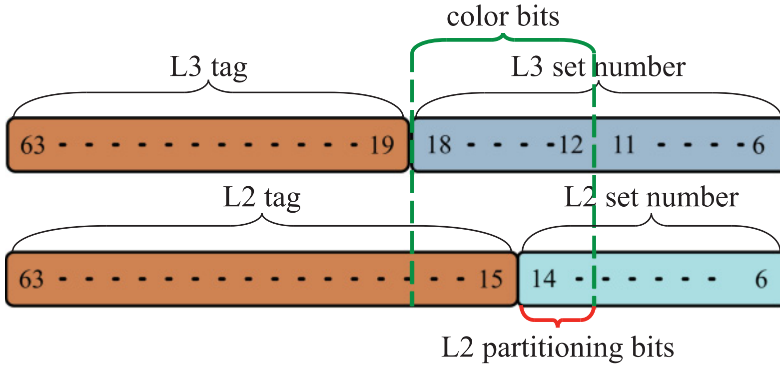

Fig. 3. Overlap of L2 set bits with color bits.

3. STATE OF THE ART

A large amount of literature is available addressing contention within the LLC, which proposes solutions of a very different nature. In summarizing the wide body of available literature, two main aspects should be distinguished: techniques and policies. On one side, the *techniques* are the mechanisms that allow control of the cache. Many works, for example, propose hardware techniques that affect the internals of the cache, often modifying the LRU priority assignment to cache lines. On the other side, the *policy* is the algorithm and metrics that drive the mechanism, deciding the actions to be applied through the mechanism. For example, a partitioning policy may gather several metrics for each core and decide the size of each cache partition according to predefined goals (such as fairness or user's QoS requirements). Both the mechanism and the policy can be implemented either in hardware or software. This distinction is fundamental in this review, and the related literature can be classified by means of these two aspects.

In the following review, we cover hardware techniques (Section 3.1) and software techniques (Section 3.2). Hardware techniques require modifications to the cache functioning, and may implement in hardware both the mechanism and the policy. However, a hardware technique can still offer interfaces to the software level in order to tune its behavior to high-level, software-defined goals. In contrast, software techniques do not require hardware modifications, and are all based on page coloring to control where application data are placed inside the cache. Section 3.3 reviews the techniques to unveil the hash function of Intel's CPUs, from which we learned the assumptions that are at the base of our reverse-engineering technique.

3.1. Hardware Techniques

Overall, hardware techniques are diverse: some change the implementation of the LRU algorithm, while others allow more explicit control over the data placement. A first mechanism for cache partitioning is called *way partitioning*, usually employing a bitmask to indicate which ways each core can access inside the cache sets. In the case of eviction, the LRU policy works only on the cache lines assigned to the core causing the eviction, so that each set is effectively partitioned among the cores. Some special-purpose architectures, such as Cavium [2004] or some prototype multicores [Cook et al. 2013], adopt this LLC partitioning mechanism, which is also finding a place in modern, commercial architectures. However, way partitioning decreases the associativity available to a core, as it allows the accessing of only a subset of the lines, offering a smaller set of candidates for eviction with respect to the case with full associativity. Several policies have been proposed to compute the number of ways for

partitioning. Among them, Utility-based Cache Partitioning (UCP) [Qureshi and Patt 2006] is the basis of several works in the literature: it defines the benefit that each application can have from receiving more ways based on the derivative of the miss-rate curve. Works such as Gupta and Zhou [2015] allow way partitioning while increasing the spatial locality with aggressive prefetching, based on samples from several sets.

Other works, instead, change the LRU policy to tackle phenomena such as thrashing or pollution. Sharifi et al. [2012] consider multithread applications and focus on fairness among cores by penalizing the core with highest IPC in favor of the others: a modified LRU policy evicts a cache line of the high-performing core in the case of miss from another. Since LRU is a history-based policy, it is unable to predict the usage of new cache lines. Therefore, low-reuse cache lines may evict high-reuse cache lines (*cache pollution*), and high-reuse cache lines can continuously evict each other (*cache thrashing*). Seshadri et al. [2012] enhance the LRU policy with information about the frequency of recently evicted lines, which is stored in a novel hardware structure and facilitates the decision regarding whether to store the incoming line or bypass it to the core. Other works attempt to predict the reuse of incoming blocks to affect the replacement policy, for example, by storing the history of incoming blocks [Li et al. 2012] or with an address-mapped table of saturating counters [Kharbutli et al. 2013]. Other works, instead, choose at runtime a certain policy based on *set dueling* [Qureshi et al. 2007], which consists of sampling the behavior of several cache sets for which a fixed policy is applied [Jaleel et al. 2010]. Recently, Khan et al. [2014] proposed two replacement policies based on the observation that the LLC should capture read-reuse patterns, while lines from write-only patterns can usually be evicted safely. Khan et al. [2014] partition each LLC set into two regions, clean and dirty regions, and use the clean region for read-only lines, while the dirty region contains written lines and is resized according to information from set dueling.

Vantage [Sanchez and Kozyrakis 2011] is a more disruptive approach based on the *z-cache* model [Sanchez and Kozyrakis 2010]. A z-cache maps the incoming data to a line by means of a hash function, achieving high associativity. Leveraging this feature, Vantage partitions the LLC by introducing two regions that capture high-reuse and low-reuse lines, respectively, and that can be resized dynamically. Wang and Chen [2014] introduce the concept of *futility*, which describes the “uselessness” of a cache line with regard to the others. A cache should always present a broad set of “futile” candidates for eviction, even in the presence of partitioning; Wang and Chen [2014] scale futility based on the insertion and eviction rates of each application’s partition.

3.2. Software Techniques

Many software techniques have been proposed to alleviate interference at the cache level. Most of these techniques assume a usual LLC addressing scheme, without a hash function (such as Intel’s Nehalem’s scheme).

Some works classify applications based on their sensitivity to co-located applications and on the contention that they cause to other applications, usually by a microbenchmark that exercises a tunable pressure in LLC and memory [Delimitrou and Kozyrakis 2013; Mars et al. 2011]. In an initial learning phase, the pressure on the memory subsystem is varied to devise the application profile,which is then used to classify the application and avoid dangerous co-locations. Pushing on this approach, Yang et al. [2013] monitor latency-sensitive applications at runtime, continuously adapting the co-location to the application phases. Other techniques directly manage the LLC and are all based on page coloring, but apply it for different goals and with different policies. For example, some tackle pollution by limiting the cache space that I/O buffers can use, either assigning a fixed number of colors [Kim et al. 2011] or identifying pages accessed sequentially (typical of I/O buffers) and mapping them to a few colors [Ding

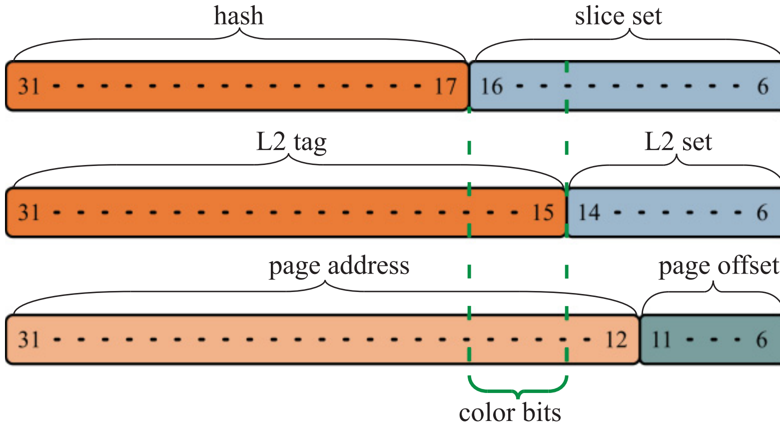

Fig. 4. Overlap of L2 set bits with color bits.

et al. 2011]. Other techniques, similarly, find polluting memory pages through the miss rate and migrate them to a small number of colors [Soares et al.2008].

To precisely partition the LLC, some approaches perform application profiling, and typically choose the number of colors for each application based on low-level metrics such as miss rate and stall rate [Tam et al. 2007]. Other works, such as Zhang et al. [2009] and Liu et al. [2016], profile the application characteristics at runtime and perform several actions, such as recoloring hot pages only. Nonetheless, this profiling can be heavy (e.g., Zhang et al. [2009] hot page recognition requires traversing the OS page table). Lin et al. [2008] evaluate several policies and metrics to guide the partitioning and find similar results in terms of overhead. Liao et al. [2014] investigate how to identify application phases by means of miss-rate curves and IPC curves, and apply recoloring to adapt partitions to phases.

Several works employ a theoretical approach to model applications' LLC characteristics. Sandberg et al. [2013] model the bandwidth usage and the performance variation due to co-location, using profile data that capture applications' characteristics and runtime phases across different time windows, validating their approach on an Intel Nehalem architecture. Brock et al. [2015] evaluate several policies of fairness with a static partitioning approach on an Ivy Bridge architecture, and provide a theoretical framework to find the optimal LLC partitioning and sharing scheme with respect to these policies. Thanks to this approach, the policies that they evaluate have general applicability.

Other works explore page coloring techniques on newer architectures. Kim et al. [2013] focus on real-time systems, developing a partitioning and sharing scheme that considers tasks with given deadlines for completion. This work starts from profiling information about the stand-alone worst-case execution time with different LLC partitions and devises a scheme for partitioning and sharing, as well as a time schedule that meets the applications' deadlines. To enforce the partitioning, Kim et al. [2013] implements a page-coloring technique on a Sandy Bridge CPU, but does not deal with the reconstruction of the hash function, thus leaving the LLC partitions spread across all the slices. Furthermore, Kim et al. [2013] employ all the bits from 12 to 17 as color bits, thus also partitioning the L2 cache (as in Figure 4). Ye et al. [2014] propose two novel recoloring policies that also consider time sharing of cores and QoS requirements, focusing instead on server environments. The first policy recolors a number of pages proportional to the memory footprint, but proves to be suboptimal since it often recolors rarely used pages. The second policy tracks page hotness by sampling the OS page

tables and remaps them to different colors to better distribute the accesses. Ye et al. [2014] also validates the proposed solutions on a low-end Sandy Bridge architecture. However, the presence of the hash-based mapping is not considered in the design phase.

Finally, page coloring also finds applicability with Virtual Machines (VMs): in this scenario, the *a priori* knowledge of the memory footprint of VMs can be used as a hint for the LLC partition size. Jin et al. [2009] use page coloring within the Xen hypervisor to show that also VMs benefit from LLC partitioning. Proceeding in this direction, Wang et al. [2012] add a dynamic recoloring mechanism that moves the most used pages, not to stop the VM during page copy.

3.3. Reverse Engineering Intel's Hash Function

Effort has been devoted to reconstructing the hash function of Intel Sandy Bridge processors, mostly for security purposes. Knowing the LLC hash function allows an attacker to perform a side-channel attack, for example, by probing hot code areas, such as cryptographic libraries and checking the load time.

Disregarding the way that they unveiled Intel's hash functions, this body of research shows that the hash is based on the XOR operator, and that it depends on the number of slices of the LLC and not on the specific architecture. With 2^n slices, the hash function consists of n different hashes that output a single bit, and that XOR certain bits. Thus, reconstructing the hash function basically reduces to finding which bits are XORed in each hash.

Hund et al. [2013] discovered the hash function of an Intel Core i7-2600 multicore by finding conflicting addresses that map to the same LLC set. However, Hund et al. [2013] manually compare the conflicting memory addresses to find repeated patterns that may hint the hash function. Irazoqui et al. [2015] adopt an analytic approach instead, solving a linear equation system in order to find the possible hashes. In their formulation, each address is a constant matrix of coefficients that multiplies a vector of unknowns, which represent the coefficients of the hash function (1 for the bit being used, 0 otherwise). Since the final solution depends on the unknown labels of the slices, Irazoqui et al. [2015] provide multiple solutions for each CPU without identifying the real solution for the CPU under test. Moreover, Irazoqui et al. [2015] handle noise by filtering addresses with intermediate access latency, assuming that the remaining ones are reliable and can thus be used for the linear system formulation. Doing so, however, requires knowledge of the CPU latencies and depends on the specific model. Finally, Wei et al. [2015] also investigates the hash function of a 4-cores and 6-core Sandy Bridge CPU, but does not provide means to represent them as formulas or as algorithms, using instead mapping tables of considerable size.

4. APPROACH AND DESIGN

With the introduction of a hash-based LLC addressing, page coloring becomes less effective in Sandy Bridge multicores. Figure 4 shows the typical memory layout for a Sandy Bridge processor, in particular, how the physical address is used to map LLC, L2, and physical memory. In Figure 4, "slice set" indicates the bits used to select the set within the LLC slice, while the rest of the bits are used to compute the hash that selects the target slice. As visible in Figure 4, the overlap between slice bits and L2 set bits leaves only 2 bits available for page coloring, thus with a granularity of only 4 partitions. Also using the L2 set bits causes partitioning of the L2 cache, which is an undesirable performance bottleneck since the L2 cache is per-core. Figure 5 shows the LLC miss rate (blue curve), the slowdown with respect to the full-LLC execution (red line) and the L2 cache miss rate (green line) of 8 applications from the SPEC CPU2006 suite [Henning 2006]. In this scenario, we used the bits 12 to 16 for partitioning, and varied the number of colors from 2 (corresponding to 0.625MB of LLC and half L2 cache)

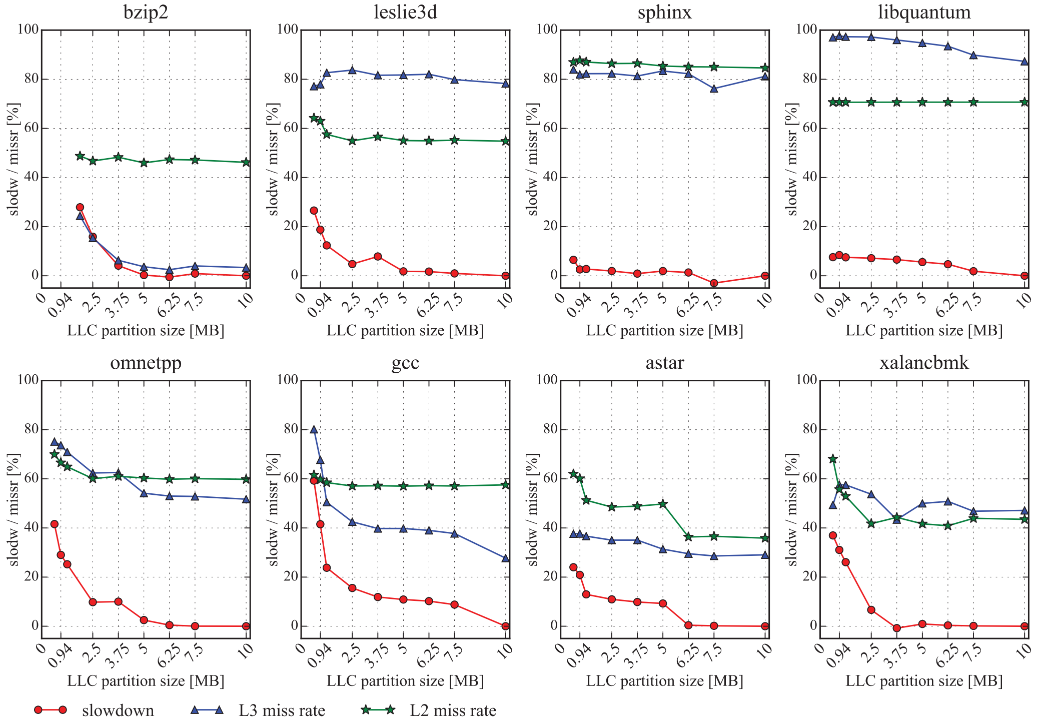

Fig. 5. Spec cache profiles when using bits 12 to 16 for partitioning.

up to 32 colors (full LLC and full L2). As a special case, bzip needs at least 1.88MB of LLC, since its memory footprint corresponds to 6 colors, thus using the whole L2 cache. In Figure 5, 5 out of 8 applications are sensitive to the LLC (we exclude bzip from this count for the aforementioned reason), and their slowdown is visibly correlated to the amount of L2 cache. For this reason, we chose not to partition the L2 cache, renouncing to the L2 set bits of Figure 4. Furthermore, since the two remaining color bits are within the slice bits and the hash function spreads accesses among all the cores, a single partition spans across 4 slices. This is undesirable, as it prevents a fine-grained control over the LLC placement and can increase the traffic on the on-chip ring bus.

To achieve a deeper control over the hash-based LLC, knowledge of the hash function is fundamental. Section 4.1 describes the assumptions and the steps to reconstruct this information using the performance monitoring features available in modern architectures, focusing in particular on Intel's Sandy Bridge architecture. With knowledge of the hash function, Section 4.2 redefines the notion of page color to adapt it to a hash-based scheme, generalizes it to the memory areas of various sizes that the buddy allocator handles, and leverages this notion to achieve an efficient and scalable implementation of a color-aware memory allocation algorithm.

4.1. Reconstruction of the Hash Function

Despite the hash function of Sandy Bridge processors being undocumented, information is available from previous work [Hund et al. 2013; Irazoqui et al. 2015; Wei et al. 2015], whose limits are shown in Section 3.3. For the hash reconstruction, we used a similar approach to Hund et al. [2013], although using a more structured flow. In our reconstruction, we initially target an Intel Xeon E5 1410 CPU with 4 cores and

6GB of RAM. From the available literature, we can confidently make the following assumptions about the hash function of our CPU:

(1) Since the CPU is a 64b CPU with four cores, the hash function has the form $h() : \{0, 1\}^{64} \longrightarrow \{0, 1\}^2$, thus is composed of two distinct scalar hash functions: $h(a) = h_2(a)h_1(a)$, where a is the input memory address. Two independent functions allow designers to combine known hashes to evenly address all four slices.
(2) The scalar hash functions are computed by XORing certain bits of the memory address. This implementation has minimum area and power overhead with current lithography, incurs minimal latency, and shows very good evenness in practice.
(3) Since our machine has 6GB of RAM, only bits 17 to 32 are used to compute the hash (as from Figure 4). The assumption of bit 17 as the lowest bit is in accordance with the literature, while the choice of bit 32 as the highest bit is due to the amount of RAM.

Although these assumptions are tailored to our specific CPU model, they can easily be generalized to any model having a number of cores of a power of 2, as previous works suggest. CPUs with a number of cores not being a power of 2 likely have nonlinear hash functions, and are left as future work.

The reconstruction consists of finding which bits each function XORs. To this aim, Hund et al. [2013] finds patterns from conflicting addresses that suggest which hash uses each bit, then finds the hash functions by manually looking at these patterns. Here, we employ a more general approach that consists of using an Integer Linear Programming (ILP) model.

Collection of conflicting memory addresses. To reconstruct h_2 and h_1, we collect memory addresses that collide with a given address a. Using the miss counter, the test accesses a (called *probe address*) first and then a sequence of addresses with distance $stride = 2^{17}B$. This stride ensures that the set that the address is mapped to is the same of a, while the slice can vary because of the unknown hash. For example, if a is mapped to set 0 of slice 1, the accessed memory positions are all mapped to set 0 of an unknown slice. After traversing l memory location (with l increasing from 1), the test reads address a and checks whether the miss count has increased. If it has, the miss has been caused by the last address read, which is recorded as a *collider*. This location has evicted a; therefore, it has the same hash. Otherwise, the test increases l and restarts reading the sequence of memory location, plus the new one at the end of the traversal. With this procedure, we collected many couples $<$ probe address, colliders $>$ in the form $< a, \{a_1, a_2, a_3, \ldots, a_n\} >$. Overall, we collected more than 2M of colliding addresses distributed among 2K probe addresses.

Hash reconstruction as a clustering problem. Thanks to assumption 1, the problem of reconstructing the hash can be reduced to a clustering problem. In the hash function h, the bits are used in three possible *configurations*:

(1) bits used only in h_1
(2) bits used only in h_2
(3) bits used in both h_1 and in h_2

For each configuration, we have to find a corresponding cluster of bits. To find bits in the same cluster, we look for colliders of the same probe having a Hamming distance of 2. Since these colliders have the same hash, we infer that the two changing bits do not change the overall hash. Therefore, given the behavior of the XOR operator, these two bits must be used in the same way (only in h_1, only in h_2, or in both), therefore belong to the same cluster (1, 2, or 3, respectively). For each couple of colliders at Hamming distance 2, we found the two different bits i and j, and counted how many times such a

couple appears across the whole dataset. This count represents the *similarity* $s_{i,j}$ of bits i and j: a high value of $s_{i,j}$ means that i and j are likely to belong to the same cluster, thus likely to be in the same configuration. Using only similarity, however, would make any clustering algorithm trivially fit all bits in the same cluster, thus maximizing the total similarity.

Thus, we need another parameter to indicate when two bits should be in separate clusters. As for similarity, we can estimate the *dissimilarity* $d_{i,j}$ of any two bits i and j to be in the same cluster. We can consider two couples probe-colliders $<a, \{a_1, a_2, a_3, \ldots, a_n\}>$ and $<b, \{b_1, b_2, b_3, \ldots, b_n\}>$ whose probes a and b have a Hamming distance 1, thus with different hashes. If any two colliders a_k and b_l have a Hamming distance 2, the two differing bits i and j must be in different clusters. Otherwise, they would cause the XOR chain of the hash to "flip" twice, resulting in the same hash (which is impossible since they have the same hash of their probes). The number of occurrences of each couple i, j is its dissimilarity $d_{i,j}$. Ideally, for any two bits i, j with $i \neq j$, it should hold that $s_{i,j} > 0$ if and only if $d_{i,j} = 0$. Nonetheless, the collected measures are affected by noise, which we attribute to the hardware performance counters not being designed for reporting a single miss. For example, if the cache miss happens right after reading the collider a_k, the miss count might be updated with some unpredictable delay and the increased count might be visible only after reading address a_{k+l}, which is mistakenly reported as a polluter. However, in our measurements, we notice the similarity and dissimilarity values to be distributed in two groups of values: one group with high count values (thousands) and one with much lower counts. We assumed this last group to be due to measurement noise; we applied a threshold to filter out these values. This is a key difference with respect to Irazoqui et al. [2015], which handles noise by filtering input data on the base of architecture-dependent latency values.

Using similarity and dissimilarity values, we can compute the best clustering by means of an ILP model. Introducing the binary variable $x_{i,j}$, which represents whether bits i and j are in the same cluster, we can write the objective function as in Equation (1), maximizing the intracluster similarity and the inter-cluster dissimilarity:

$$\text{maximize} \sum_{i=17}^{31} \sum_{j=i+1}^{32} [x_{i,j} \times s_{i,j} + (1 - x_{i,j}) \times d_{i,j}]. \tag{1}$$

For the clustering to be meaningful, we also have to force the transitivity property of clustering: if bit i is in the same cluster of bit j and j is in the same cluster of k, then bits i and k must also be in the same cluster. Therefore, we add the constraints in Equation (2):

$$\begin{aligned} &\forall i, j, k \mid 17 \leq i < j < k \leq 32 : \\ &x_{i,k} \geq x_{i,j} + x_{j,k} - 1, \qquad x_{i,j} \geq x_{i,k} + x_{j,k} - 1, \qquad x_{j,k} \geq x_{i,j} + x_{i,k} - 1^{.} \end{aligned} \tag{2}$$

Using an ILP solver, we found the optimal solution to be the following three clusters of bits:

$$\begin{aligned} c_1 &= 18, 25, 27, 30, 32 \\ c_2 &= 17, 20, 22, 24, 26, 28 \\ c_3 &= 19, 21, 23, 29, 31. \end{aligned}$$

These clusters are the same as those in Hund et al. [2013], although the multicore is different.

Configuration choice via LLC access latencies. The next step is determining to which *configuration* a cluster corresponds: for example, c_1 can correspond to h_1 (configura-tion 1), c_2 to h_2 and c_3 can be the configuration of common bits; or any other combination,

	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
h2		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕											
h1	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕											

Fig. 6. Hash function of Intel Xeon E5 1410.

each one corresponding to a different hash function. Thus, there are six possible hashes. To find the one of our multicore, we leveraged the different access latencies from cores to slice: if a core (numbered from 0 to 3) accesses the slice of its same number, the latency must be minimal, because each core is directly connected to its slice and does not pass through the ring bus. To obtain this measurement, we wrote a simple test that accesses sequential memory regions and measures the access latency. This measurement is possible by using a specific performance counter available on our multicore that measures the memory latency in the case of L2 miss. Throughout the measurements, the minimum value (that appeared consistently) was 18 cycles, which we assumed to be the one indicating access to the local LLC slice. Finally, we repeated the test on each core i with all possible hashes, and stored, for each core, the hashes that return i when the access latency is 18 cycles. At the end, only the hash function in Figure 6 makes correct predictions for all cores, and is assumed to be the real hash function.

Results with another multicore. To validate our reconstruction methodology, we applied it to a machine with an 8-core Intel Xeon E5 E5-2690 with 256GB of RAM. After collecting roughly 2.5M memory addresses, from the clustering phase we obtained the following 7 clusters:

$$c_1 = \{24, 17, 20, 28, 33\}, c_2 = \{31, 34, 19, 37, 23\}, c_3 = \{32, 25, 18\}, c_4 = \{26, 35, 22\},$$
$$c_5 = \{27, 36, 30\}, c_6 = \{21\}, c_7 = \{29\},$$

which correspond to the 7 possible cases (shared among all hashes, shared between any two or exclusive). With these 7 cases, we should theoretically test the predictions for all cores of $7! = 5040$ possible hashes, corresponding to the permutations of the clusters. However, we observe that the clusters have different sizes, and the bigger clusters are unlikely to be shared by all of the hashes. This would limit the "entropy" among hashes, potentially causing access bottlenecks to few slices. On the opposite side, c_1 and c_2 are likely to be shared and not reserved for single hashes since this would cause the one hash to have different entropy from the two using c_1 and c_2. Thus, these clusters are likely to be shared by couples. These assumptions limit the number of candidate hashes to 432 (3 candidates for the cluster shared by all cores, 4 candidates for the cluster shared by couples, and 3 candidates for the reserved clusters), which makes the approach feasible in a reasonable time. For CPUs with a higher number of cores, this approach can be very time-consuming, as it scales with the number of permutations of the clusters. However, future manycore CPUs will probably not rely on a ring-based architecture, and will have a completely different structure. Therefore, we consider scalability issues to remain limited to a reasonable scale.[1] After testing, c_5 emerges as the shared cluster, and the three hash functions of the processor are

$$h_1 = c_5 \oplus c_1 \oplus c_3 \oplus c_7, \quad h_2 = c_5 \oplus c_1 \oplus c_2 \oplus c_6, \quad h_3 = c_5 \oplus c_2 \oplus c_3 \oplus c_4,$$

where we use the $\oplus$ operator between clusters to indicate the XOR of all bits of the clusters.

[1]It is possible to use branch and bound or backtracking strategies to find the final hash, constraining the research space over and over with missing addresses whose mapping slice is known (e.g., by checking the latency). However, this optimization is out of the scope of this article, and is left for future work.

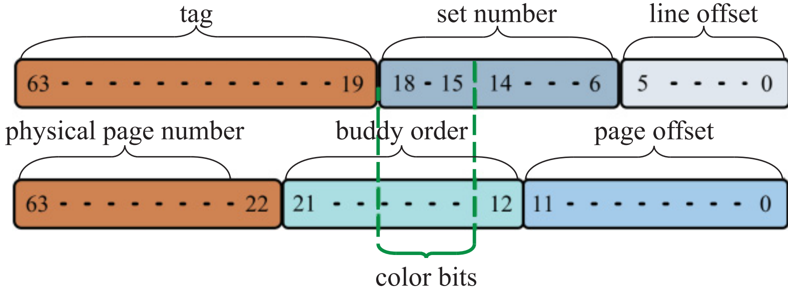

Fig. 7. Bits of buddy addresses forced to 0 and overlapping with color bits.

4.2. Definition of Color and Structure of the Color-Aware Buddy Allocator

With the knowledge of the hash function, we can redefine the color of a page by padding the two original color bits with the two hash bits so that the color of a page with the address $a = a_{63} \ldots a_0$ becomes $c(a) = h_2(a)h_1(a)a_{16}a_{15}$. Due to this redefinition, the number of available LLC partitions is 16, a sufficient granularity. To make the lookup for a page of a specific color constant, we split the list of pages used by the buddy algorithm into 16 sublists, one per color. In this way, the algorithm can select a sublist in constant time, and remove the first page from the list (still in constant time). However, the notion of color is defined only for pages, while the buddy algorithm manages physical memory in chunks, called *buddies*, of different sizes.

Each buddy is characterized by two parameters: the order, typically from 0 to 10, indicates its size, so that a buddy of order i is composed of 2^i physically contiguous pages; the address of the buddy is the address of the first memory page of the buddy, and is always aligned to the size of the buddy (its lower i bits, from bit 12 to bit $12+i-1$, are 0). In the case in which no pages of a desired color are present, the algorithm should split a buddy of order 1 and check which page has the desired color. If, for example, the desired color is $c = 0011_2$, the algorithm should split a 1-order buddy, whose bit 12 is forced to 0 due to the memory alignment of buddies. Therefore, the color bits are not affected by the alignment, and the algorithm should look for a 1-order buddy of color c. To make this search constant, the list of 1-order buddies should be split into 16 sublists, as for the pages. As from Figure 7, this holds until order 3, as there is no overlap between color bits and bits forced to 0 in the buddy address. In the case in which the algorithm has to split a buddy of order 4, bit 15 is forced to 0, and the color of the buddy to look for is 0010_2: then, the algorithm should return the second half, whose bit 15 is 1. In this case, 8 configurations of the color bits are possible, since bit 15 is 0; thus, the list of order 4 should be split into 8 sublists. For higher colors, the available configurations of color bits further decrease by a factor of 2: 4 configurations for order 5, 2 for order 6, and 1 for orders 7 and higher.

To deal with the buddy lookup in a unique way, we generalize the notion of color to buddies of any order through the definition of the *mcolor*. For our testbed multicore, the *mcolor* can be defined, in base 2, as

$$mcolor(a, i) = \begin{cases} h_2(a)h_1(a)a_{16}a_{15} & \text{if } 0 \le i \le 3 \\ h_2(a)h_1(a)a_{16} & \text{if } i = 4 \\ h_2(a)h_1(a) & \text{if } i = 5 \\ (h_2(a)\ XOR\ h_1(a)) & \text{if } i = 6 \\ 0 & \text{otherwise.} \end{cases}$$

This definition models our previous example and considers only the bits that can vary at each order, so that it can be used to query which sublist to look in at each order. It

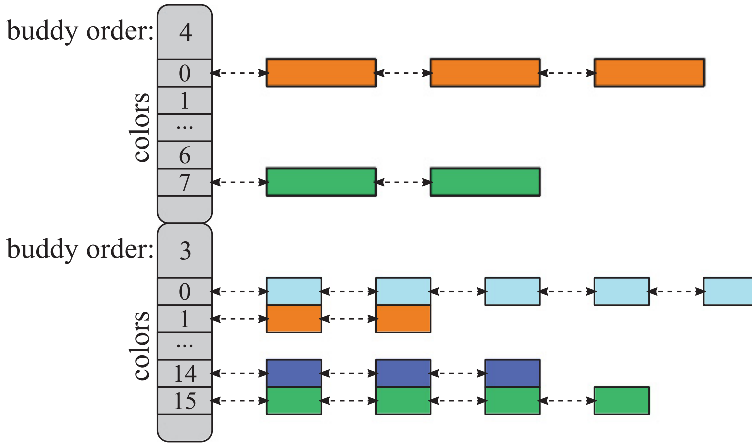

Fig. 8. Data structure of the color-aware buddy algorithm.

allows the computation of the mcolor of each buddy in order to insert it to the proper sublist when it is freed. In particular, order 6 represents a special case, as it defines the *mcolor* as the negated equality between h_1 and h_2. This definition is due to the role of bit 17, which is forced to 0 in buddies of order 6. Figure 6 shows that this bit is used in both hash functions h_1 and h_2: therefore, if the hashes are equal, the hashes of the two 5-order sub-buddies will also be equal, since bit 17 either flips both the hashes (if it is 1) or none (if it is 0). Similarly, if the hashes are different, the hashes of the sub-buddies will be different. If, for example, the desired color is 1010_2, it is possible to select either a 6-order buddy with hash 10_2 or one with hash 01_2 (both having different hashes). After the split, the algorithm will choose in the former case the first sub-buddy (whose bit 17 is 0, hence not flipping) and in the latter case the second sub-buddy, whose bit 17 being 1 will cause both hashes to flip. Finally, for orders greater than 6, the buddy contains all the possible colors, which map to 0.

Since the buddy allocator receives requests of pages of specific colors, we need to efficiently compute the mcolor from the color in order to perform a fast lookup. Similar to the definition of the *mcolor*, we can map the requested color $c = c_3c_2c_1c_0$ to the *mcolor* of a given order i by means of the following function:

$$mcolor_lookup(c, i) = \begin{cases} c_3c_2c_1c_0 & \text{if } 0 \leq i \leq 3 \\ c_3c_2c_1 & \text{if } i = 4 \\ c_3c_2 & \text{if } i = 5 \\ (c_3 \; XOR \; c_2) & \text{if } i = 6 \\ 0 & \text{otherwise.} \end{cases}$$

With these definitions, it is possible to design a color-aware allocation algorithm based on the buddy allocator that avoids lookup, maintaining efficiency and scalability.

Figure 8 sketches the structure of the modified buddy algorithm. In place of a single list of buddies for each order, we modified the data structure at the base of the buddy algorithm to have one list per *mcolor* for each order. Thus, at orders 0 to 3 there are 16 lists, order 4 has 8 lists, order 5 has 4, order 6 has 2 and orders from 7 on have a single list. Using one list per *mcolor* allows the algorithm to select a buddy of the desired *mcolor* in constant time. Consequently, also the procedures to select a colored page and to insert one are modified. Algorithm 1 shows the procedure to select a page of a desired color: if the page is present in the color list, it is returned immediately; otherwise, a buddy of higher order is split. The procedure *SplitBuddy* computes the target *mcolor*

ALGORITHM 1: Split and Select Procedures

```
1  globaldata:      list_head  buddies[MAX_ORDER][MAX_COLORS]
2
3  procedure SplitBuddy(order ord, mcolor mcol): buddy
4          buddy first, second, current
5          mcolor firstc, secondc, targetc
6
7          if ord == MAX_ORDER return nil
8
9          targetc = McolorLookup(mcol, ord)
10         if ListIsEmpty(buddies[ord][targetc])
11                 current = SplitBuddy(ord+1,mcol)
12                 if current == nil return nil
13         else current = RemoveHead(buddies[ord][targetc])
14
15         first,second = ComputeHalves(current)
16         firstc = Mcolor(first, ord)
17         secondc = Mcolor(second, ord)
18         if firstc != targetc
19                 Swap(first, second)
20                 Swap(firstc, secondc)
21         InsertHead(buddies[ord-1][secondc],second)
22         return first
23 end procedure
24
25 procedure SelectPage(color col): buddy
26         if ListIsEmpty(buddies[0][col])
27                 return SplitBuddy(1,mcol)
28         else
29                 return RemoveHead(buddies[0][col])
30 end procedure
```

(line 9), looks for the buddy of correct *mcolor*, eventually splitting a higher order buddy via a recursive call (lines 10–13), then splits the buddy in two halves (line 15) and checks which half has the target *mcolor*, adding the other free half to the list of free buddies (16–21).

Similarly, the procedure to insert a freed buddy (Algorithm 2) is based on the definition of *mcolor*, which is computed according to the definition (line 7). Then, the insertion algorithm checks whether the "twin" buddy is also free: in case it is, it coalesces them into a single buddy of higher order and recursively calls the insertion procedure (lines 11–13); otherwise, it inserts the free buddy into the proper list (line 15).

Thanks to the definition of *mcolor* and, consequently, of the function *mcolor_lookup*, the splitting and insertion procedures do not perform any lookup to find a requested color, but execute in constant time independently from the number of buddies and, ultimately, from the size of the physical memory, maintaining the scalability and efficiency of the original algorithm.

The proposed allocator has been implemented in Linux 3.17, modifying the existing implementation of the buddy allocator, which also requires considering the hierarchical memory distribution consisting of memory nodes and *zones*[2], as well as other heuristics to control memory fragmentation. For the purpose of a realistic implementation, we

[2]In Linux terminology, a node corresponds to a physical memory node, while a zone is a partition of a node from which kernel subsystems allocate preferably: for example, the first B are the *DMA zone*, since old DMA controllers handling a physical address of 24 bits can manage buffers only in this zone.

Table I. Selected SPEC CPU2006 Tests

Test	Input
libquantum	control
gcc	g23
omnetpp	omnetpp
leslie3d	leslie3d
xalancbmk	t5
sphinx	ctlfile
astar	rivers
bzip2	text

ALGORITHM 2: Insertion Procedure

```
 1 globaldata: list_head buddies[MAX_ORDER][MAX_COLORS]
 2
 3 procedure InsertBuddy(buddy b, order ord)
 4         buddy twin
 5         mcolor mcol
 6
 7         mcol = Mcolor(b, ord)
 8         twin = GetTwinBuddy(b,ord)
 9         if ord < MAX_ORDER-1 AND BuddyIsFree(twin)
10                 RemoveFromList(buddies[ord][Mcolor(twin,ord)])
11                 b = CoalesceBuddy(b, twin, ord)
12                 InsertBuddy(b, ord+1)
13                 return
14         else
15                 InsertHead(buddies[ord][mcol],b)
16 end procedure
```

integrated our design into the existing codebase, modifying the routines that manage each zone. On top of those routines, the algorithms that select the node and the zone work as usual. To provide users with a suitable interface, a new *cgroup* [Menage 2004] has been implemented to expose the LLC partitioning capabilities. This interface allows users to manually create LLC partitions by specifying the colors of each partition and the applications that use the partition.

5. EXPERIMENTAL RESULTS

This section discusses the experimental setup to evaluate the proposed solution in Section 5.1 and presents a first evaluation in Section 5.2, showing how it is possible to control the LLC usage of stand-alone single-core applications by means of LLC partitioning. Discussing the behavior of the selected applications, the section also devises application mixes to run in co-locations, which are evaluated in Section 5.2. Finally, Section 5.4 evaluates the LLC partitioning of co-located multithreaded benchmarks.

5.1. Methodology and Testbed

To evaluate the effectiveness of the proposed solution, we selected 8 benchmarks from the SPEC CPU2006 suite [Henning 2006]. These benchmarks have been selected because they have diverse profiles with respect to LLC usage and are mostly sensitive to the LLC size. Table I shows the selected applications along with their input sets, which are those that cause the longest runs. In a first phase, each application is profiled stand-alone, and the number of colors is varied from 1 to 16. Section 5.2 shows the application profiles, describing its behavior with LLC partitions of any size. For LLC

partitioning to be beneficial, these applications should ideally have the same profiles in co-location with others. Section 5.3 defines several workloads, in which a *target* application is pinned to a core and isolated in LLC and 3 other applications are co-located on the other cores and contend for the rest of the LLC, acting as *polluters* to the application that benefits from isolation in LLC. Also, with these setups, the target applications are profiled with several partition sizes. The aim of a variable partition size is to allow the final user to select the size that guarantees the requested QoS, which can be expressed, in the case of the SPEC benchmarks, as the maximum tolerated slowdown with respect to the stand-alone execution.

The testbed machine has the following characteristics:

—Intel Xeon E5 1410, with 4 cores running at 2.80GHz
—6GB RAM DDR3 at 1333MHz
—L1 instruction and data caches of 32KB each
—L2 unified cache of 256KB
—LLC of 10MB, composed of four slices
—Page size of 4KB
—16 colors

To evaluate the benefits of LLC partitioning, we disabled advanced features that affect the LLC performance, such as TurboBoost, the prefetchers, and the power-saving features of the kernel.

5.2. Application Profiles

Figure 9 shows the profiles of the applications: the horizontal axis reports the size of the LLC partition (each color corresponds to 0.625MB of LLC); the vertical axis reports the percentage of slowdown with respect to the execution with 10MB of LLC (red lines) and the miss rate (blue lines). The plots for gcc and bzip2 start from 1.88MB of LLC due to their memory requirements: since partitioning the LLC limits the available physical memory (due to assigning less colors), the input sets of these two applications do not fit in less than 800MB, corresponding to a 1.88MB LLC partition.

Figure 9 also shows the behavior of applications with a "classical" page-coloring scheme with dotted lines: in this scenario, the partitioning mechanism is unaware of hashing ("w/o hash" in the legend) and uses bits 15 to 18 for partitioning, as in Figure 3 (in which we do not use L2 partitioning bits). Figure 9 shows the effectiveness of our solution in controlling the usage of the LLC, also highlighting which applications are more sensitive to the amount of cache. The bars, which represent the standard error of the mean, in most of the plots are barely visible, showing that applications with a regular memory access pattern have a predictable behavior with varying LLC partitions. In contrast, sphinx shows considerable variability, which is due to its unfriendly access pattern (caused by a search heuristic) and to the application itself repeatedly loading inputs (small speech samples). The figure shows that hash-unaware partitioning always performs better than hash-aware partitioning: in some cases, the hash-unaware miss rate at 1.25MB is comparable to the hash-aware miss rate at 5MB or more. This phenomenon is due to the underlying functioning of the LLC: although hash-unaware partitioning uses bits 15 to 18, only bits 15 and 16 are effective. Bits 17 and 18 are used in the hash, which also varies depending on the higher bits that change from page to page. Therefore, when the allocator chooses a page only on the basis of bits 15 to 18, its hash is uniformly distributed on the whole range, resulting in more LLC space allocated than the space requested. For example, setting only color 0 keeps bits 15 and 16 to 0 while the slice varies, resulting in the allocation of a quarter of *each* slice. Thus, hash-unaware curves in Figure 9 have larger plateaus than hash-aware curves. Instead, based on hash-aware profiles, applications have been classified

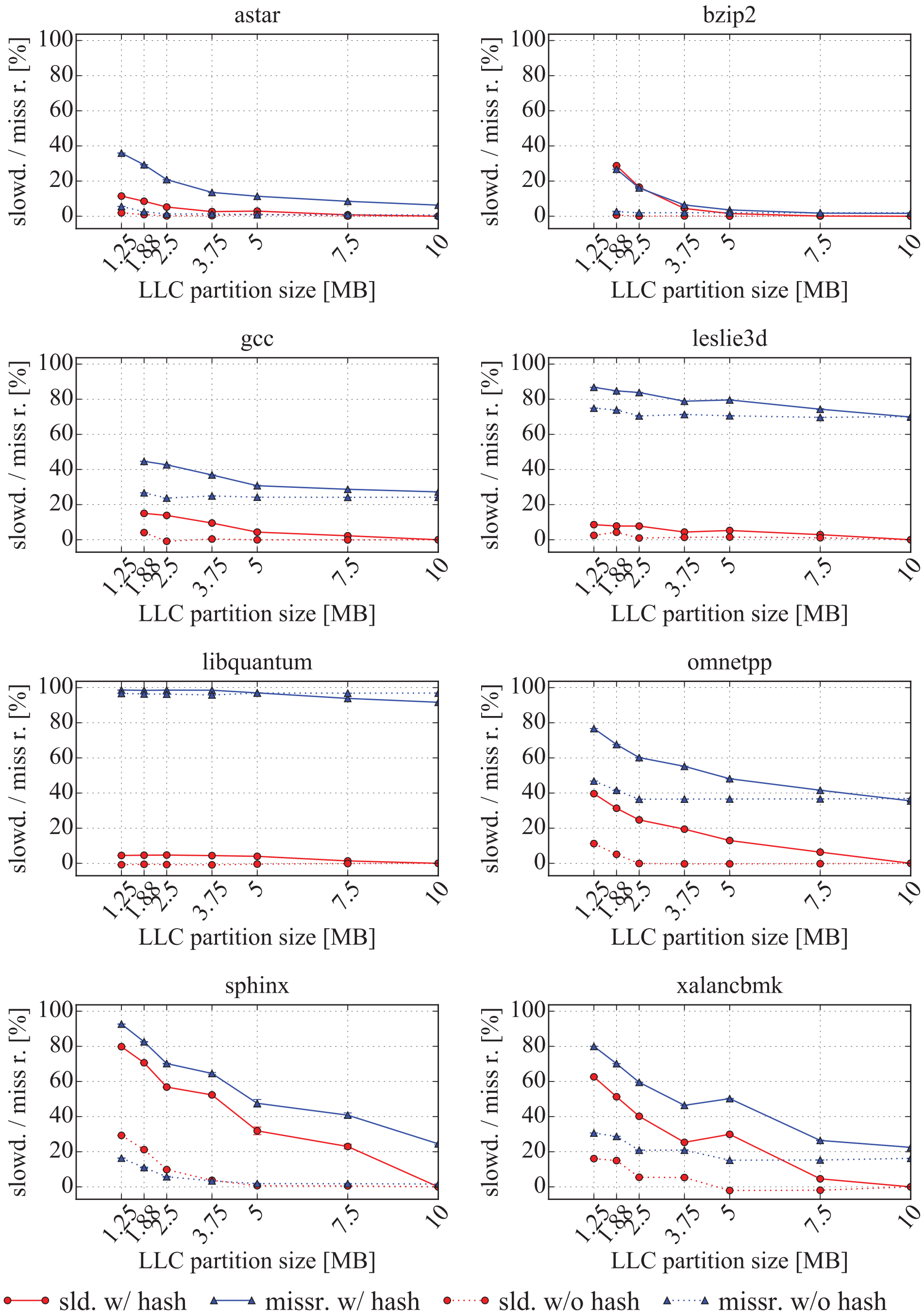

Fig. 9. Application profiles with different cache partitions, with slowdown (“sld”) and miss rate (“missr”) hash-aware partitioning (“w/ hash”) and hash-unaware partitioning (“w/o hash”).

Table II. Classification of Applications

Classification	Applications
sensitive	bzip2, omnetpp, xalancbmk, sphinx
insensitive	libquantum, leslie, astar, gcc

Table III. Test Workloads

Workload	Target	Polluters
W1	bzip2	xalancbmk, leslie3d, gcc
W2	omnetpp	sphinx, libquantum, astar
W3	xalancbmk	omnetpp, libquantum, gcc
W4	sphinx	bzip2, leslie3d, astar

according to their sensitivity to the partition size. Applications whose slowdown with the least amount of cache is equal to or greater than 30% are defined to be *sensitive*, while the others are *insensitive*. Table II shows how the eight reference benchmarks are classified.

5.3. Co-location Profiles

To evaluate isolation capabilities in co-location, we devised one workload for each sensitive application, called *target* application, which runs co-located with three other applications chosen randomly, called *polluters*. To have a diverse mix, the first polluter is chosen from the sensitive applications while the other two polluters are insensitive applications. Table III shows the four workloads, with the target and the polluters. Throughout all the tests, we run the entire application with the biggest input, and we immediately restart the polluters whenever they terminate.

Figure 10 presents the results from the workload runs: the continuous lines show the slowdown (red) and the miss rate (blue) of the target application with hash-aware partitioning, while the dotted lines show slowdown and miss rate with hash-unaware partitioning. Furthermore, the dashed black lines show the harmonic means of the polluters' slowdowns with hash-aware partitioning. To avoid the polluters from swapping to disk, the target applications receive at most 3.75GB of RAM, and the three polluters share the remaining 2.25GB. Because of the memory partition effect, 3.75GB of RAM corresponds to the 6.25MB of LLC. The profiles show that hash-aware partitioning is more effective in controlling the performance for a varying LLC partition, while with hash-unaware partitioning, the targets suffer from higher contention in the LLC (as from the miss rate line). Hash-unaware partitioning is more effective in the case with 1.88MB of LLC, since the LLC space devoted to the target is higher than 1.88MB due to bits 17 and 18 not limiting the LLC space. With higher amounts of LLC, bits 15 and 16 can assume any value, so that any set within any slice is possible, and this mechanism is ineffective.

Comparing Figure 10 to Figure 9, the targets show a less regular behavior with respect to the stand-alone profiles. The polluters have, on average, less variations, since two of them are cache-insensitive, but exercise a noticeable pressure on the LLC, affecting the target performance. In particular, W1 has a small LLC footprint, and partitioning is able to keep the application's data in the LLC. W2 has a larger memory footprint and a cache-friendly access pattern, and benefits from having large LLC space. W3 and W4 have irregularities that are due to the I/O activity of the polluters. Since the targets (xalancbmk and sphinx) have higher duration than the polluters, which are immediately restarted, multiple I/O bursts occur during the execution of the target. During these bursts, the kernel I/O subsystem employs a page of any available color (kernel colors are always unrestricted to prevent hotspots in the kernel execution), mapping buffers to the colors allocated to the target. This phenomenon, exacerbated by

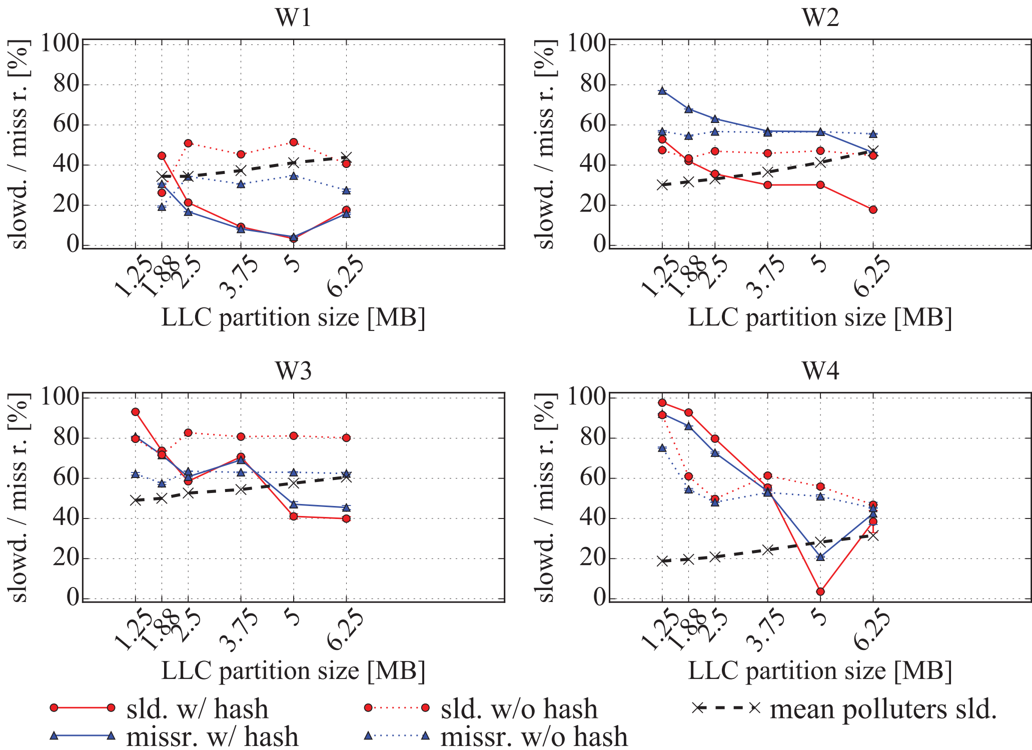

Fig. 10. Profiles of the workloads in Table III, with different cache partitions: continuous lines refer to the target with hash-aware partitioning, dotted lines refer to the target with hash-unaware partitioning, and dashed lines refer to the polluters.

Table IV. Test Workloads with Limited I/O Activity

Workload	**Target**	**Polluters**
X1	bzip2	xalancbmk, leslie3d, *astar*
X2	omnetpp	sphinx, libquantum, *xalancbmk*
X3	xalancbmk	omnetpp, libquantum, *leslie3d*
X4	sphinx	bzip2, leslie3d, *sphinx*

the limited availability of memory, is well visible in W4, whose target (sphinx), has the longest execution time. Therefore, we devised 4 other workloads, reported in Table IV, whose applications have similar execution times. These workloads derive from those in Table III by replacing the polluter having the least duration with the highlighted one. For example, in the case of W4, all the polluters have very different durations from the target sphinx, and the only possible replacement was another instance of sphinx in lieu of astar. Figure 11 plots the resulting profiles similarly to Figure 10 (continuous lines for the target with hash-aware partitioning, dotted lines for the target with hash-unaware partitioning, dashed black lines for the polluters' slowdown with hash-aware partitioning), showing more regular curves that are closer to the stand-alone profiles.

5.4. Multithreaded Co-location Profiles

To further evaluate the effectiveness of our hash-aware partitioning scheme, we perform a similar evaluation with multithreaded applications from the PARSEC 3.0 suite

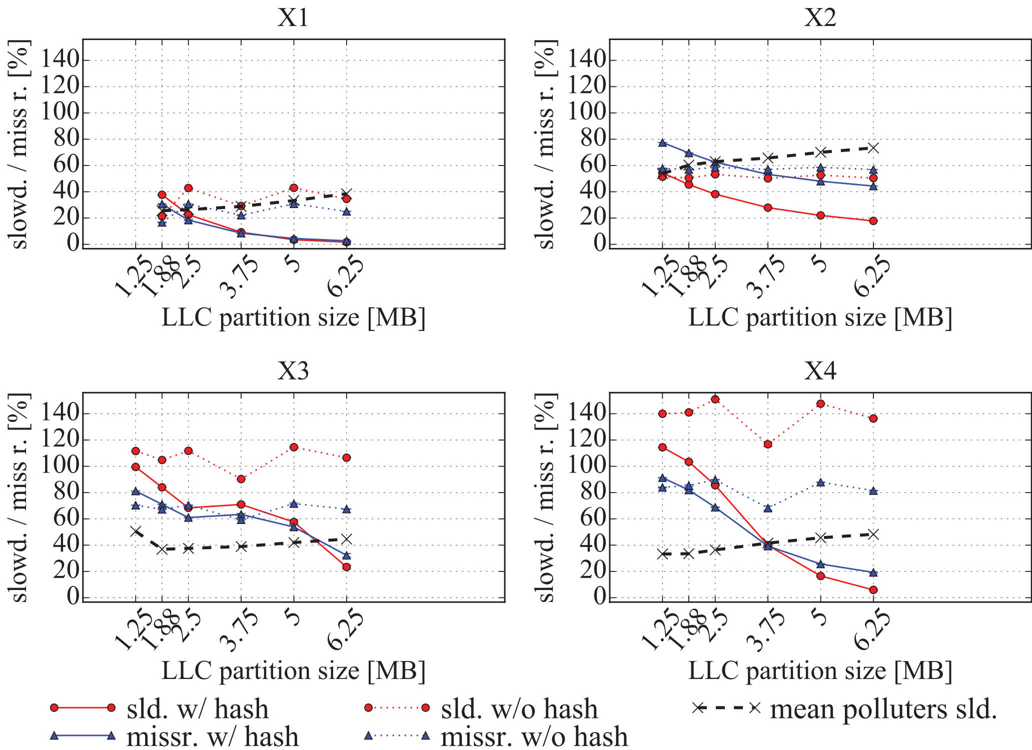

Fig. 11. Profiles of the workloads in Table IV, with different cache partitions.

Table V. PARSEC Co-located Workloads

Workload	Target	Polluter
P1	bodytrack	swaptions
P2	ferret	facesim
P3	freqmine	raytrace
P4	vips	blackscholes

[Bienia et al. 2008]. For this evaluation, we co-locate two applications from the PARSEC suite: for each pair, the first application is selected as target, while the second acts as polluter and contends the LLC to the target. Both applications perform complete runs and have 4 threads each to maximize contentiousness (with standard OS time-sharing of CPU cores), and are given the PARSEC native input; to deal with different execution times, the polluter application is immediately restarted as soon as it terminates. The application pairs, named from P1 to P4, are shown in Table V. Figure 12 shows the profiles of the co-located pairs, in which the dotted lines represent the stand-alone execution (not present in Figure 9) and the continuous lines the execution in co-location. Here, while the miss rate values in co-location are close to those of the stand-alone, the slowdown is significantly impacted by contention on computational bandwidth and on memory access.

6. CONCLUSIONS AND FUTURE WORK

This work proposes a technique for LLC partitioning based on page coloring that is able to work also on modern hash-mapped caches. The proposed design aims at maintaining the scalability and efficiency of the Linux Buddy allocator, while allowing the selection

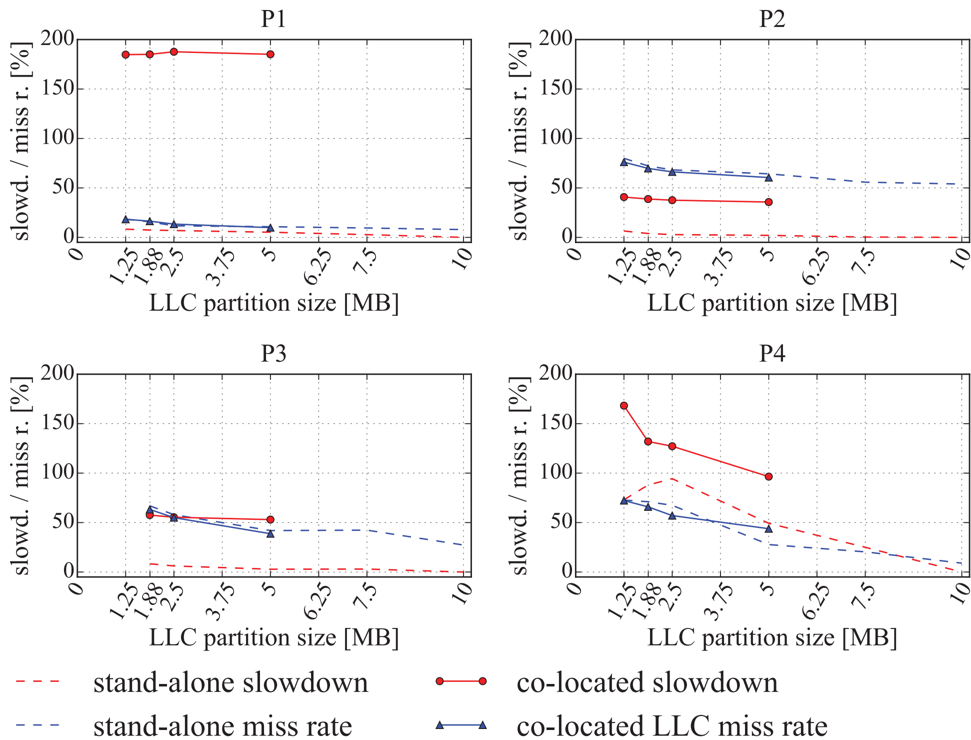

Fig. 12. Profiles of the workloads in Table V, with different LLC partitions.

of memory pages of any given color. All the work is based on the knowledge of the LLC hash function, which is reconstructed by means of widely available performance counters. The technique presented to reconstruct this information is based on assumptions that are reasonably valid across multiple architectures. Therefore, the validation of our approach to more architectures is possible future work, as well as the evaluation of the allocator design on a broader range of configurations in order to test its efficiency and scalability capabilities.

In the testbed environment, this technique was effective in controlling the usage of the LLC of selected applications running, even if some limitations emerged. However, while this technique is hindered by pollution due to OS buffers, orthogonal research [Soares et al.2008] provides a solution that can be integrated to mitigate this issue.

In completing this work, a policy that drives our partitioning technique is the most natural extension. This policy can, for example, also take in account the on-chip traffic among slices and further enforce performance isolation, possibly using recoloring policies to adapt to a dynamic workload.

REFERENCES

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 72–81. DOI:http://dx.doi.org/10.1145/1454115.1454128

Brian K. Bray, William L. Lunch, and Michael J. Flynn. 1990. *Page Allocation to Reduce Access Time of Physical Caches*. Technical Report. Stanford, CA, USA.

Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP'15)*. IEEE Computer Society, Washington, DC, 749–758. DOI:http://dx.doi.org/10.1109/ICPP.2015.84

Cavium. 2004. Octeon processors family by Cavium Networks. Retrieved December 2, 2016 from http://www.cavium.com/newsevents_octeon_cavium.html.

Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 308–319. DOI:http://dx.doi.org/10.1145/2485922.2485949

Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News* 41, 1, 77–88.

Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. 2011. SRM-buffer: An OS buffer management technique to prevent last level cache from thrashing in multicores. In *Proceedings of EuroSys*.

Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. 2010. Managing contention for shared resources on multicore processors. *Communications of the ACM* 53, 2, 49–57. DOI:http://dx.doi.org/10.1145/1646353.1646371

S. Gupta and H. Zhou. 2015. Spatial locality-aware cache partitioning for effective cache sharing. In *2015 44th International Conference on Parallel Processing*. 150–159. DOI:http://dx.doi.org/10.1109/ICPP.2015.24

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4, 1–17. DOI:http://dx.doi.org/10.1145/1186736.1186737

R. Hund, C. Willems, and T. Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (SP'13)*. 191–205. DOI:http://dx.doi.org/10.1109/SP.2013.23

Intel Corp. 2015. *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. Technical Report. Retrieved December 2, 2016 from http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. Cryptology ePrint Archive, Report 2015/690. Retrieved December 2, 2016 from http://eprint.iacr.org/.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 60–71. DOI:http://dx.doi.org/10.1145/1815961.1815971

Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li. 2009. A simple cache partitioning approach in a virtualized environment. In *Proceedings of ISPA*.

S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jimenezz. 2014. Improving cache performance using read-write partitioning. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 452–463. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835954

M. Kharbutli, M. Jarrah, and Y. Jararweh. 2013. SCIP: Selective cache insertion and bypassing to improve the performance of last-level caches. In *IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT'13)*. 1–6. DOI:http://dx.doi.org/10.1109/AEECT.2013.6716445

Hyoseung Kim, Arvind Kandhalu, and Ragunathan (Raj) Rajkumar. 2013. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*. IEEE Computer Society, Washington, DC, 80–89. DOI:http://dx.doi.org/10.1109/ECRTS.2013.19

JongWon Kim, Jinkyu Jeong, Hwanju Kim, and Joonwon Lee. 2011. Explicit non-reusable page cache management to minimize last level cache pollution. In *Proceedings of ICCIT*.

Kenneth C. Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. DOI:http://dx.doi.org/10.1145/365628.365655

Oded Lempel. 2011. 2nd Generation Intel Core Processor Family: Intel Core i7, i5 and i3. Retrieved December 2, 2016 from http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.911-Sandy-Bridge-Lempel-Intel-Rev%207.pdf.

Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu Cheng. 2012. Optimal bypass monitor for high performance last-level caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 315–324. DOI:http://dx.doi.org/10.1145/2370816.2370862

Xiaofei Liao, Rentong Guo, Danping Yu, Hai Jin, and Li Lin. 2014. A phase behavior aware dynamic cache partitioning scheme for CMPs. *International Journal of Parallel Programming* 1–19.

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of HPCA*.

L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu. 2016. Rethinking memory management in modern operating system: Horizontal, vertical or random? *IEEE Transactions on Computers* 65, 6, 1921–1935. DOI:http://dx.doi.org/10.1109/TC.2015.2462813

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 248–259.

Paul Menage. 2004. Control Group Linux documentation. Retrieved December 2, 2016 from https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 381–391. DOI:http://dx.doi.org/10.1145/1250662.1250709

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of MICRO*.

Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of MICRO*.

Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of ISCA*.

A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. 2013. Modeling performance variation due to cache sharing. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 155–166. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522315

Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. 2012. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of PACT*.

Akbar Sharifi, Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2012. Courteous cache sharing: Being nice to others in capacity management. In *Proceedings of the 49th Annual Design Automation Conference*.

Livio Soares, David Tam, and Michael Stumm. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 258–269.

David Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Proceedings of WIOSCA*.

Ruisheng Wang and Lizhong Chen. 2014. Futility scaling: High-associativity cache partitioning. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 356–367.

Xiaolin Wang, Xiang Wen, Yechen Li, Yingwei Luo, Xiaoming Li, and Zhenlin Wang. 2012. A dynamic cache partitioning mechanism under virtualization environment. In *Trust, Security and Privacy in Computing and Communications (TrustCom'12)*. IEEE, 1907–1911.

Zhipeng Wei, Zehan Cui, and Mingyu Chen. 2015. Cracking Intel Sandy Bridge's cache hash function. *arXiv preprint arXiv:1508.03767*.

Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 607–618.

Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. ACM, New York, NY, 381–392. DOI:http://dx.doi.org/10.1145/2628071.2628104

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multi-core cache management. In *Proceedings of EuroSys*.