Make Page Coloring More Efficient on Slice-based Three-Level Cache

Haifeng Li*†, Tianyue Lu*†, Yuhang Liu*, Mingyu Chen*†‡
*University of Chinese Academy of Sciences, Beijing, China
†State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
‡Peng Cheng Laboratory, Shenzhen, China
{lihaifeng19b, lutianyue, liuyuhang, cmy}@ict.ac.cn

***Abstract*—On modern multi-core machines, page coloring has been used to alleviate the competition at Last Level Cache (LLC). However, the latest development of CPU architecture has brought new issues to page coloring. Firstly, in the case of three-level cache, previous works about page coloring did not discuss the impact on L2 cache of color allocation and the competition for L2 cache is not considered concurrently under hyper-threading. In addition, as the last level cache structure is changed from shared to slice-based and undocumented hash function is applied, page coloring is more complex and slice information is also not fully utilized.**

This paper presents solutions to these issues. Firstly, by making small changes to the traditional page coloring, the problem that page coloring may waste L2 cache is alleviated. At the same time, we rethink the vertical allocation of L2 cache and LLC in page coloring under hyper-threading, and discuss the impact of color allocation on programs, especially those with different sensitivity to L2 cache and LLC. Finally, we make full use of slice information and propose Partial Conflict Color (PCC). At the same time, we also propose a fast method to obtain PCC. Experiments show that using PCC can improve system performance when the number of colors is insufficient.

I. Introduction

With the development of chip technology, system designers place more processor cores on a single chip, and hyper-threading technology is proposed to improve thread/task level parallelism. The Last Level Cache (LLC) design of recent processors has changed. Specifically, the Intel LLC is divided into multiple slices, and cache lines are mapped to different slices by an undocumented hash function. As the processor structure is changed, the method of cache partition has also changed. Last Level Cache partitioning is a popular solution to reduce or eliminate interference across applications co-running on multi-core processors. A few approaches have been proposed, which can be roughly divided into three categories: way partitioning, page coloring and the combination of way partitioning and page coloring.

Way partitioning is achieved by assigning a subset of the cache ways to different cores, thus divvying up the last-level cache space. Way partitioning has little cost on hardware implementation, but it reduces the associativity of each partition, adding cache misses [12]. The second method is page coloring. By modifying OS, the pages with same color bits in their page numbers (the page frame ID overlaps the cache index) are grouped into one color. Pages of different colors occupy different LLC. This approach has been adopted in real operating systems. The third method of partitioning cache is SWAP (Set and Way Partitioning) [10], which provides hundreds of fine-grained cache partitions. Among the three methods, page coloring is the most widely used methods, which requires no hardware modification and only needs to modify the OS. Therefore, a lot of researches are based on page color. Such technique was adopted by commercial OS such as Solaris, FreeBSD, netBSD, and Windows NT [21]. However, changes in the architecture of Intel processors cause the following issues:

Firstly, as L2 cache and LLC are both set-associative, color bits (the shared bits between a physical address page frame ID and LLC index, address bits 12-16) also have impact on the allocation of L2 cache. The specification of the cache hierarchy in Intel(R) Xeon(R) CPU E5-2620 v2 is shown in Table I. A subset of color bits can be called L2 color (address bits 12-14). One L2 color corresponds to 12.5% L2 cache capacity, and four L3 colors correspond to one L2 color. Improper L3 color allocation leads to the waste of L2 cache, which is more likely to occur in dynamic color allocation. Meanwhile, the performance of L2 cache-sensitive programs can be badly reduced.

The second issue is as follows: in the case of hyper-threading, two processes running on the same physical core share L2 resources. Cache contention and conflicts were not addressed simultaneously at multiple cache levels.

The third issue is that the number of colors is limited. This limitation is worse when caches are indexed using hashing, which is common in LLCs of modern processor. For example, a non-hashed LLC has 128 colors, but a hashed LLC with the same capacity consisting of four slices can only support 32

TABLE I
Intel(R) Xeon(R) CPU E5-2620 v2 Cache Specification

Cache Level	Size	Ways	Sets	Index-bits(address bits)
L1	32 KB	8	64	11-6
L2	256 KB	8	512	14-6
LLC-Slice	2.5 MB	20	2018	16-6

This work is supported by National Key Research and Development Plan of China 2017YFB1001602, NSFC (National Science Foundation of China) No. 61772497 and No. 61521092, State Key Laboratory of Computer Architecture Foundation under Grant No. CARCH2601.

978-1-7281-2583-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPADS47876.2019.00051

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

colors [21].

In order to alleviate the above issues, we propose the following methods:

To address the first issue of the insufficient utilization of L2 cache caused by page coloring in three-level cache structure, we use the method proposed by Yuval Yarom [3] to divide a color into two slice colors based on the slice information. Two slice colors belong to the same set and occupy the same part of L2 cache. In this way, when assigning a color to a program, we can assign two slice colors that are not in the same set, making the L2 cache available to the program twice as much as before. At the same time, when assigning colors to a process, we try to assign colors that do not share L2 cache. Ultimately, the process allocates L2 cache as much as possible without changing the proportion of L3 cache. Experiment results show that IPC is increased by 20.12% in the best case and increased by 4.74% on average.

To address the second issue, we analyze the situation that programs with different sensitivities to L2 and L3 running on different hyper-threading of the same physical core and propose that L2 and L3 cache should be considered comprehensively. Experiment results show that IPC is increased by 6.9% in the best case and increased by 5.5% on average.

To address the third issue, we propose the partial conflict color (PCC) by using the slice information of each cache line, which upgrades the original 32 colors to 256 partial conflict colors. Meanwhile, we sum up the PCC formula and propose a simple method to extend the formula. Extensive experiments show that the overall performance of the system can be improved by using PCC when the number of colors is insufficient. Experiment results show that IPC is increased by 7.91% in the best case and increased by 3.38% on average.

The rest of this paper is organized as follows: Section II provides background and Section III introduces the motivation. Section IV describes the methods and experiments we proposed to solve the problem of insufficient utilization of L2 cache. Section V discusses the vertical allocation of L2 and LLC under hyper-threading and the corresponding case study. Partial conflict color and related experiments are described in section VI, followed by related work and conclusions.

II. Background

A. Memory Hierarchy

With the development of multi-core processor technology, there is a higher demand for memory subsystems. However, memory access time is much slower than processor. Therefore, processor designers have designed cache to bridge the speed gap between the processor and the memory.

Cache is used to speed up data access. It is often divided into private cache and shared cache. Each core has its own private cache (L1 cache), which contains data cache and instruction cache. In addition, some processors have added a layer of private cache, L2 cache, such as Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz. The shared cache is usually L2 cache (in two-level cache structure) or L3 cache, also known as Last Level Cache (LLC), which is shared between the cores of the processor.

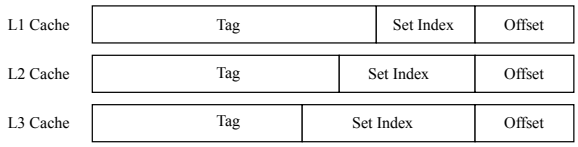

Fig. 1. The Structure of Physical Address in three-level Cache Structure

Cache reads and writes at the granularity of 64 B cache line. Each cache line is mapped to a cache set. The number of cache lines that each set can store is called the way. The three-level cache has different set and way numbers and use different bits for addressing (Fig. 1). When the processor accesses a specific address, it first searches for the data corresponding to its tag in the set according to the set index. If the data is in the corresponding set, then the access is responded by L1 cache, which we call L1 cache hit. Otherwise, L1 cache miss occurs. The processor continues to look for data in L2 cache according to L2 set index and tag. If found, it is L2 cache hit, otherwise it is L2 cache miss. After L2 cache miss occurs, the processor continues to repeat the above steps in LLC. If LLC miss occurs, the processor will continue to look for data in DRAM.

B. Slice-based Last-Level Cache

Starting with the Sandy Bridge microarchitecture, Intel redesigned the LLC by dividing the LLC into multiple slices. Each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events (Fig. 2). The processor uses an undocumented hash function to determine which slice the address maps to, and L3 index to determine the set of the address. On this processor, the number of bits of L3 index decreases, which results in fewer color numbers available for page color. At the same time, the undocumented hash function has caused great difficulties in the research of processor security.

There have been many studies on hash function. Some researchers probe the memory to find which memory location maps to each cache set [1]. Ralf Hund [2] rebuilds hash

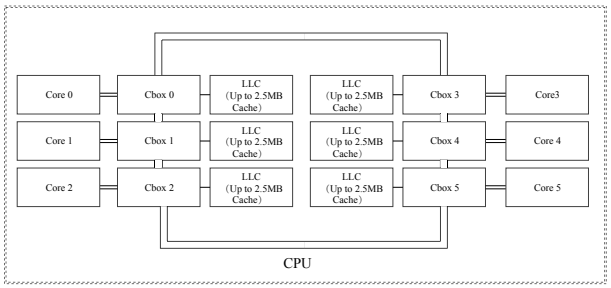

Fig. 2. Example of Intel Xeon Processor E5-2600 v2

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

function on a 4-core machine and proved that set index cannot determine which slice the address would fall on. Clementine Maurice [4] uses performance counter to get the access number of per slice and summarize the hash function of 2, 4, 8 cores processors. Wei [5] uses HMTT [14] to grab the physical address trace and analyze the addresses that collide with each other in the same set to get addresses that are on the same slice, but this method cannot get the specific slice ID of the physical address. Yuval Yarom [3] uses the principle that processors access different slices in different time, obtains the mapping relationship between address and slice on the 6 core machine, and reconstructs hash function. When they get the hash formula, they find that they can divide an old color with the same set into two new colors without cache conflict, which is called slice color.

C. Cache Partition

Cache partition usually refers to the partition of shared L2 or L3 cache. These cache partition methods have different optimization objectives, including performance [15], [16], [18], fairness [15], [19], and QoS (Quality of Service) [17], [19].

Page coloring is a widely used method of partition cache. In most processors, a shared LLC is physically indexed and set associative. The processor searches for data in cache according to the physical address, which is divided into a tag, a set index and an offset. The offset bits are used to determine the specific offset of data in the cache line. The set index is used to select the specific set. Tag is used to check whether the current cache line is hit or miss. The overlapping bits of page frame number (PFN) and set index are called color bits. The operating system assigns pages of different colors to different processes to achieve cache isolation. The formula for calculating the number of colors is :

$$\text{number of colors} = \frac{\text{cache size}}{\text{number of ways} \times \text{page size}}$$

However, some recent architectures have introduced cache slices [7], as described in section II-B, this new structure

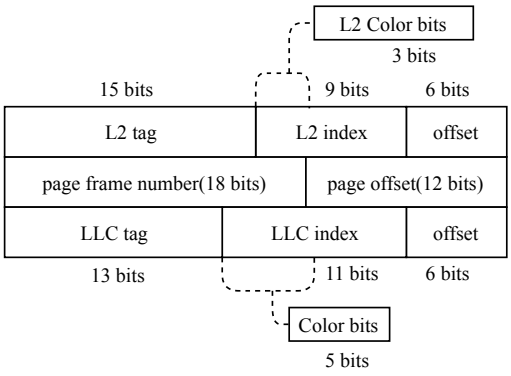

Fig. 3. Example of physical address mapping for page coloring, corresponding to Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz architecture used in this study.

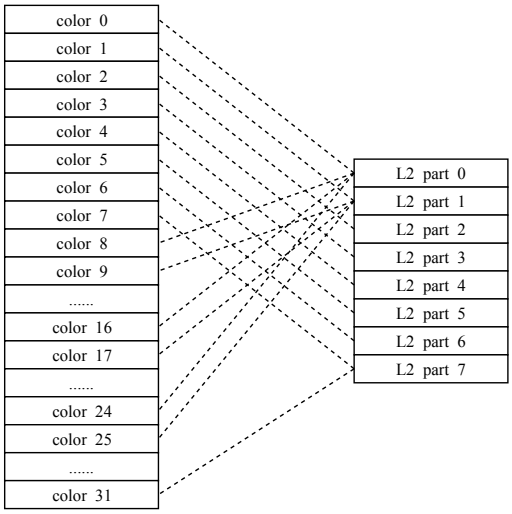

Fig. 4. The relationship between color and L2 cache

brings many difficulties and new problems to page coloring, which we will discuss in detail in the next section.

III. Motivation

A. Page Coloring Leads to Inadequate Utilization of L2 Cache

In the three-level cache structure, L2 and L3 cache are set-associative. In Fig. 3, L2 Color bits (the shared bits between a physical address page frame number and L2 index) affect L2 allocation and are a part of the color bits.

The relationship between color and L2 cache is shown in Fig. 4. When four colors are assigned to a process, the L2 cache that the process use varies greatly. For instance, the system assigns 4 colors (0, 1, 2, 3) to a process, the process can use 50% of the L2 cache. However, if the system allocates colors (0, 8, 16, 24) to the process, the process can only use 12.5% of the L2 cache. An inappropriate color allocation can degrade the L2 resources available to the program, resulting in the poor performance of the program. This situation is more serious for L2-sensitive programs. In order to make the process use 100% of L2 cache, at least eight colors must be assigned to the process. Otherwise, L2 cache will be wasted.

Previous experimental environments for page coloring were two-level cache structures, and none of them considered that L2 cache occupancy of a process varies greatly, when assigning the same number of colors to the program.

B. Considering the Influence of Page Coloring on L2 and L3 Cache under Hyper-threading

The allocation of page coloring will cause changes in L2 cache occupancy, so when allocating color, not only the number of colors should be considered according to the process's sensitivity to LLC, but also the number of "types" of colors should be considered according to the process's sensitivity to L2 cache. If two processes run on different physical cores, they will not compete with L2 cache. But if they run on different

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

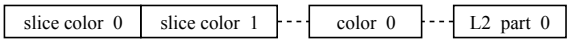

Fig. 5. Association among slice color, color and L2 cache

hyper-threading of the same physical core, they compete for L2 cache. When two programs with different sensitivity to L2 and L3 are executed concurrently, we need to choose the best strategy by considering both the cache hierarchy information and application characteristics.

C. Inadequate Use of Slice Information

There have been many previous studies on CPU with slice structure, Yuval Yarom [3] proposed that on Xeon E5-2430, 32 page colors could be expanded to 64 slice colors by using slice information. However, a non-hashed LLC has 128 colors [20], we need to continue mining slice information to increase the number of colors and mitigate the impact of the problem of assigning a small number of colors to multiple programs.

IV. Make full use of L2 cache in page color

A. How to Make Full Use of L2 Cache?

Under the existing schema, unreasonable color allocation leads to the waste of L2 cache and reduces the performance of L2-sensitive programs. We propose two ways to alleviate this problem.

First, colors should be grouped, which means if two colors correspond to the same part of L2 cache, they should be in the same group. Our experimental platform consists of 32 colors, which should be divided into eight groups, each with four colors. In static or dynamic page coloring allocation, OS should assign colors belonging to different groups to programs. When the OS recycles part of the color of the program, it should make the process occupy as much L2 cache as possible according to the distribution of the color in eight groups. In this case, each program needs an array to record the number of colors belonging to eight groups. In this way, colors are evenly distributed in different groups and L2 cache can be maximized and evenly utilized.

In addition, we use the method proposed by Yuval Yarom [3] to divide a color into two slice colors. In Fig. 5, each color can be divided into two slice colors that do not conflict on LLC but conflict on L2 cache. When a process needs to apply for a color, the OS can assign it two slice colors instead. Consequently, the L2 cache available to the process is doubled without changing the LLC allocation of the process.

B. Experiment Platform

We implemented a prototype system for a 64-bit Centos 4.7 with kernel version 3.10.93. All the experiments were conducted using the SPEC CPU2006 benchmark suite on Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz processor and 64GB of RAM. The L3 cache was shared amongst the 6 cores (12 hyper-threads) of the processor. The processor has 256KB 8-way set-associative L2 private cache per physical core. As a result, there were 32 page colors available in the system with

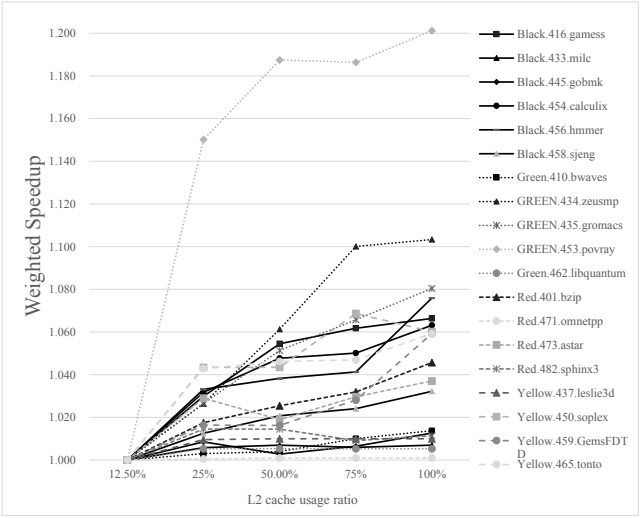

Fig. 6. Sensitivity of spec programs to L2 cache.

4KB page size. We modified the buddy memory system of Linux kernel to assign pages of different colors to processes.

C. The Effectiveness of Improving L2 Usage of Programs

In order to control the effect of L3 cache on program performance, we fixed the number of colors to four to ensure that the size of LLC used by the process is constant. With four colors, processes use up to 50% of L2 cache and 12.5% of L2 cache at least. By using the method described in section IV-A four colors can be divided into eight slice colors, so that processes can use up to 100% of L2 cache.

Lin, J [6] divides SPEC programs from high to low into four categories according to their sensitivity to LLC: red, yellow, green and black. We select several kinds of programs from each class to analyze their sensitivity to L2 cache.

We counted the sensitivity of 19 SPEC programs to L2 cache, including 4 red programs, 4 yellow programs, 5 green programs and 6 black programs. Fig. 6 shows the sensitivity of 19 programs to L2 cache. IPC is increased by 20.12% in the best case and increased by 4.74% on average. There are 7 programs in 0% - 1.5%, 3 programs in 3%-5%, 7 programs in 5%-10% and 2 programs in more than 10%. The most L2-sensitive procedures are: 3 green programs (over 8%), 3 black programs (6.3%-7.5%) and 3 yellow programs (about 6%).

By allocating slice colors belonging to different groups, we can make the best use of L2 cache and reduce the waste of L2 cache due to unreasonable allocation in the process of static or dynamic allocation of color.

V. Rethinking the vertical allocation of L2 and LLC in page color

A. Classification of Programs with Different Sensitivity to L2 and L3 Cache

In the case of hyper-threading, multiple processes compete for L2 and L3 cache simultaneously. It is important to rethink

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

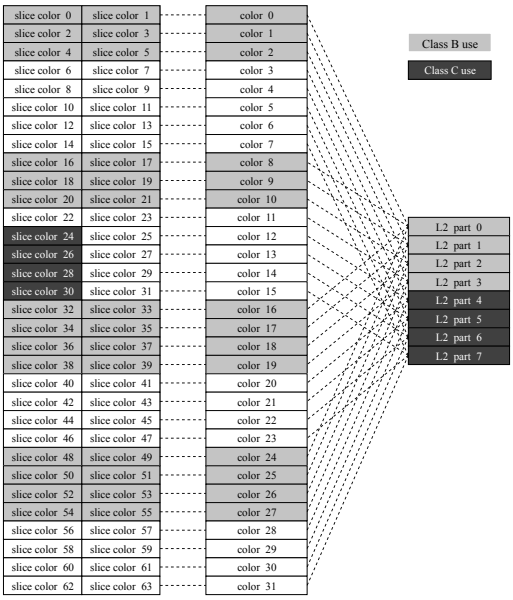

Fig. 7. the best L2 cache allocation when the LLC ratio is 14 to 2 for Class B and Class C.

the vertical allocation of L2 and L3 cache in page coloring. Depending on whether the program is sensitive or insensitive to the L2 cache and L3 cache, the programs can be divided into four categories.

When the sensitivity of one layer of cache is different between two processes, OS needs to maximize the overall performance by changing the "type" of color. In this study, we do not focus on designing a dynamic scheduling algorithm, mainly for static allocation which explores the impact of the system performance for different class processes running on hyper-threading.

B. Case Study of Vertical Allocation of L2 Cache and LLC in Page Color

In section V-A, we classify programs into four categories. When two processes are running in different hyper-threading

TABLE II
WHEN COLOR IS ALLOCATED CONTINUOUSLY, THE RATIO OF L2 CACHE TO L2 CACHE CONFLICT FOR TWO PROCESSES

	Class B	Class C	conflict
15:1	100%	12.5%	12.5%
14:2	100%	25%	25%
13:3	100%	37.5%	37.5%

TABLE III
USING THE METHOD OF SECTION IV-A, THE RATIO OF L2 CACHE TO L2 CACHE CONFLICT FOR TWO PROCESSES

	Class B	Class C	conflict
15:1	50%	25%	0%
14:2	50%	50%	0%
13:3	50%	75%	25%

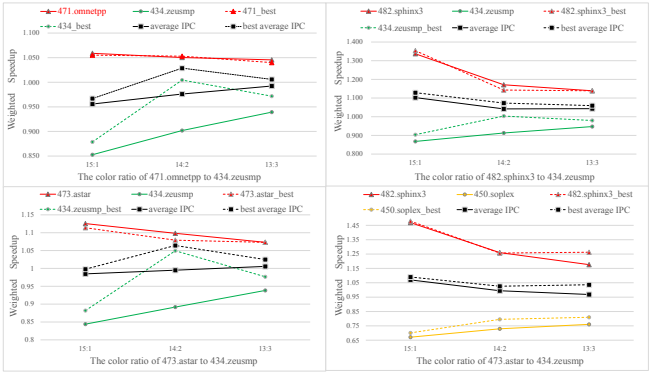

Fig. 8. The performance change chart of continuous allocation and optimal allocation for two programs with different LLC ratios, dotted line is the optimal allocation.

concurrently, we mainly discuss the following scenario: Class B (not sensitive to L2 cache, but sensitive to LLC) and Class C (sensitive to L2 cache, but not sensitive to LLC) run together The experimental platform is consistent with the description in section III. In the experiment, the two processes use a total of 16 colors. Baseline is that two processes share 16 colors, which means they share about 8MB LLC and 256KB L2 cache. We calculate IPC for class B and C programs at the color ratio of 15 to 1, 14 to 2 and 13 to 3. Because fewer colors for L3 cache-sensitive programs can make performance lower than baseline, we will not discuss them.

Under each color ratio, we do two experiments, one is to allocate continuous colors to the processes, and the other is to use the method of section IV-A. In the second experiment, the two programs occupy the same proportion of LLC, while class B programs occupy as little L2 cache as possible, and class C programs occupy as much L2 cache as possible. For example, when the color ratio is 14 to 2, colors are allocated continuously. Process B allocates color 0 to 14, and process C allocates color 15. B uses 100% L2 cache and C uses 25% L2 cache. The L2 cache used by C is part of the L2 cache used by B. We use the method of section IV-A to change the L2 cache proportion of two programs without changing their L3 cache proportion, as shown in Fig. 7. In this case, the LLC usage size of the two processes remains unchanged, but each accounts for half of the L2 cache.

When the LLC ratio is 15 to 1, 14 to 2 and 13 to 3, the L2 cache occupancy ratio and conflict ratio of the two programs in continuous allocation of color and optimal allocation are shown in Table II and Table III

According to the conclusion in section IV-A we use class B programs: 471. omnetpp, 473. astar, 482. sphinx3. Class C programs are 434. zeusmp, 450. Soplex. Fig. 8 shows that if the allocation of L2 and L3 cache is considered comprehensively, the performance of class B programs (insensitive to L2 cache and sensitive to L3 cache) is not affected much, but the performance of class C programs (insensitive to L2 cache and insensitive to L3 cache) is improved a lot. As a result,

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

the overall performance has been improved, with a maximum increase of 6.9% and the average increase of 5.5%.

Experiments show that when doing page coloring, L2 and L3 cache allocation need to be considered concurrently. In multi-core processors with hyper-threading, if programs with different sensitivity to L2 and L3 cache are placed in hyper-threading of the same physical core, the cache resources can be fully utilized and the overall performance can be greatly improved by allocating slice color reasonably.

VI. Make full use of slice information in page color

A. Partial Conflict Color

Pages of different colors are distinguished by set index. Yuval Yarom [3] prove that in the case of knowing the slice id of each cache line, it can be found that there is no cache conflict between the two pages of the same set. This allows using different colors for pages that share the same cache-set index address bits, increasing the number of supported colors. In other words, a color can be divided into two non-conflicting slice colors. However, the slice information is not fully utilized. By making full use of the slice information, we can divide 1 color into 8 partial conflict colors. At the same time, we sum up the PCC formula and propose a simple method to extended the formula.

Our experimental environment is the same as described in section IV-A. Cache-set index bits limits page coloring to only 32 different colors. First, we use the performance counter to get the slice information of $4 \sim 5$G physical memory and define the collision rate of two pages, that is, the number of cache lines with the same offset of two pages falling on the same slice divided by the number of cache lines contained in the page. For instance, in Fig. 9, page A and B have 64 cache line size blocks and the number of blocks with the same offset and slice ID is 6. Therefore, the collision rates of page A and B is $6/64 = 9.375\%$.

We divided pages with a collision rate of less than 65% into one group, and eventually, pages with the same set index were divided into eight groups, called partial conflict color (PCC). The collision distribution between partial conflict color is shown in Fig. 10. The black part represents the L3 cache occupied by PCC. Each lattice accounts for one-sixth of the L3 cache occupied by a slice color. Two PCCs occupy the same column lattice, which means that they will have L3 cache replacement.

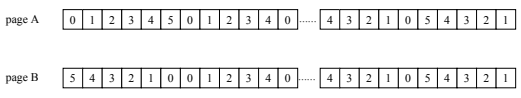

Fig. 9. Slice ID for each cache line size block of pages A and B.

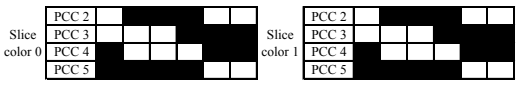

Fig. 10. The relationship between partial conflict color (PPC) and slice color.

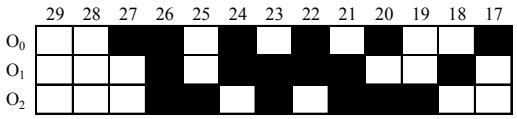

Fig. 11. Formula for calculating partial conflict color in $4 \sim 5$G of memory. Shaded boxes show address bits that are XORed to compute each output bit.

TABLE IV
The Corresponding Relations of Partial Collision Colors in Different Memory Areas

4-5G partial conflict color	0	1	2	3	4	5	6	7
5-6G partial conflict color	2	3	0	1	6	7	4	5

It can be found that pages with partial conflict colors 0, 1, 6, 7 and pages with partial conflict colors 2, 3, 4, 5 do not replace each other in LLC. These two groups of partial conflict colors are called slice colors. The collision rate of two partial conflict colors belonging to the same slice color may be 0, 1/3 or 2/3. Experiments show that this conclusion applies to all sets. At the same time, we summarize the formulas for obtaining these colors. Calculate the physical address offset relative to 4G, a total of 30 bits, address [0:29]. Offset (6 bits) and LLC index (11 bits) are not involved in the calculation, so low 17 bits are not used as input. We take the high 13 bits, address [17:29] as the input. The specific formula is shown in Fig. 11.

B. A Fast Method to Extended the Formula of Partial Conflict Color

It is not enough to know the formula of partial collision color of $4 \sim 5$G physical address space. We must extend the formula to more digits. Although we can obtain slice information of larger address space by previous methods to derive partial conflict color formula of 1TB memory space, it takes at least 1024 times as long as obtaining 1GB memory formula. Therefore, a fast method for expanding partial conflict colors need to be found.

According to the existing formulas, we judge that when the number of address bits increases, the new number of address digits will increase to the new formula in the XOR way. We apply the formulas derived from $4 \sim 5$ G memory to $5 \sim 6$ G, dividing pages of same set into eight new partial conflict colors. We use performance counter to record the number of slices accessed by a new partial collision color. The relationship between old and new conflict colors is shown in

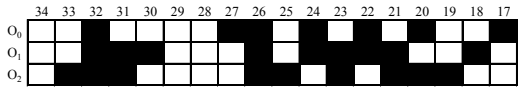

Fig. 12. Formula for calculating partial conflict color in $4 \sim 36$G of memory. Shaded boxes show address bits that are XORed to compute each output bit.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

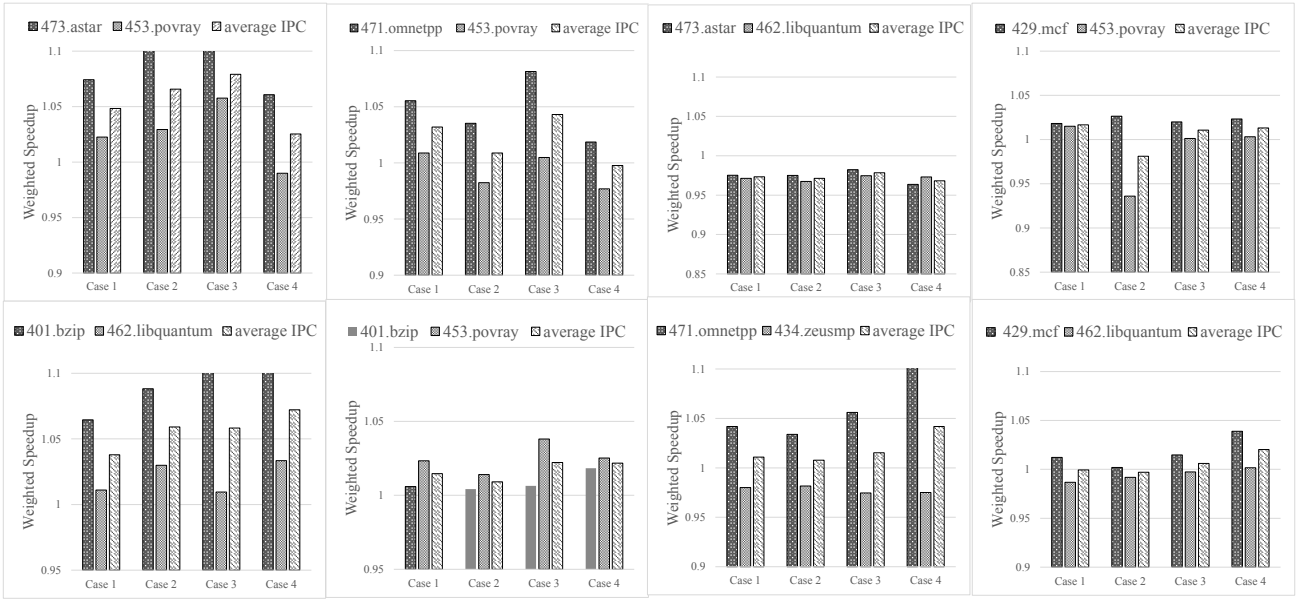

Fig. 13. Performance of cache-sensitive programs and cache-insensitive programs under different cache partitioning strategies.

Table IV. Therefore, the new bit address [30] only participates in the calculation of output [1].

By repeating the above steps, we expand the input to 17 bits and get the formula of partial color from 4 $\sim$ 36 G of physical memory. The result is shown in Fig. 12.

In this way, the formula address number can be increased by one bit each time only by measuring the slice number of pages with performance counter. It reduces the time spent on acquiring slice information of the entire address space and the time spent on analysis.

C. The Effectiveness of Partial Conflict Color

In complex systems, many threads or processes allocate color, and the number of remaining colors is insufficient. We discuss the case where only 2 colors are left and 2 programs need to be allocated.

When processes are sensitive or insensitive to LLC, colors can be divided into two programs in a ratio of 1 to 1, which is the baseline. We mainly discuss the following situations: program A is sensitive to LLC , program B is insensitive to LLC.

We compare evaluation results of four configurations (Table V):

TABLE V
LLC PROPORTION AND CONFLICT PROPORTION OF FOUR DIFFERENT CACHE PARTITION METHODS

Type	A	B	Cache Conflict
case 1	75%	25%	0%
case 2 (PCC)	100%	25%	25%
case 3 (PCC)	91.6%	13.6%	8.33%
case 4 (PCC)	100%	12.5%	12.5%

1) Case 1 (cache partition using slice color). In previous sections, we know that a color can be divided into two non-conflicting slice colors. Therefore, we assign three slice colors to the cache-sensitive program and one slice color to the cache-insensitive program. Cache-sensitive programs occupy 75% of the cache, and cache-insensitive programs occupy 25% of the cache. There is no cache replacement between them.

2) Case 2 (cache partition using partial conflict color). Each color can be divided into eight partial conflict color (PCC). The cache-insensitive program takes up two PPCs, PCC0 and PCC6. The cache-sensitive program takes up one full color and the remaining six PPCs. The cache occupancy ratio and conflict ratio of the two programs are shown in Table V.

3) Case 3 (cache partition using partial conflict color). The ache insensitive program takes up two PPCs, PCC0 and PCC7. The cache-sensitive program takes up one full color and the remaining six PPCs. The cache occupancy ratio and conflict ratio of the two programs are shown in Table V.

4) Case 4 (cache partition using partial conflict color). The cache-insensitive program takes up one PPC, PCC0. The cache-sensitive program takes up one full color and the remaining seven PPCs. The cache occupancy ratio and conflict ratio of the two programs are shown in Table V.

We count the IPC of the two programs in a fixed period of time. Fixed time period is 4 minutes. And we find that making the fixed period longer does not change the final conclusion. We divide the IPC of other cases by the IPC of the baseline and compare the partial conflict with the partition.

Fig. 13 show that using PCC is effective when the L3 cache sensitive program A and the L3 cache insensitive program B are co-running, so that the overall IPC of the system is improved. By discovering a balance point that allows cache-

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.

sensitive programs to take up more cache and less conflict with the two program caches, the IPC of the cache-sensitive program is improved and the IPC of the cache-insensitive program does not decrease too much. In the best case, 401 and 462 run together, the system IPC is increased by 3.5%. System IPC is increased by 1.71% on average. In the case of 471 and 434 running together, the IPC of cache-sensitive program is increased by 6.7% in the best case, and the IPC of cache-sensitive program is increased by 2.77% on average.

VII. RELATED WORK

Most experimental platform for page coloring research is a two-level cache structure and L2 cache is a shared LLC. They did not consider the impact of page coloring on L2 cache in a three-level cache structure. At the same time, the allocation strategy of page coloring under hyper-threading is not considered. Although some studies have used a three-level cache processor, they have not explored the impact of page coloring on L2 cache.

Lin, J [6] transferred page coloring from the simulator to the real system. He experimented on Dell PowerEdge 1950 machine with a shared L2 cache. Livio Soares [8] used page coloring to store unused pages in pages with specified colors to improve performance with a shared L2 cache. Ye, Y. [9] presented a memory management framework called COLORIS on a quad-core Intel Xeon E5506 2.13GHz processor and 8GB of RAM. The L3 cache was shared amongst the 4 cores of the processor. Wang, X [10] combined page coloring with way partitioning to make cache partitioning more fine-grained to improve system performance and experimented on ThunderX CN8800 [11]. With a shared, 16 MB L2 cache. Nosayba El-Sayed [12] experimented on Xeon D-1540 cores (Broadwell) with 256KB private per-core, 8-way set-associative L2 cache and 12 MB, shared, 12-way set-associative L3 cache, but the proposed method is based on way partition, using Intel's CAT technology. Alireza Farshin [13] used slice structure to move data closer to the core to improve performance. Although slice structure is used, it is not applied to page color.

VIII. CONCLUSION

We explored the issue that page coloring can lead to L2 cache waste on CPUs with slice structure. By allocating colors belonging to different groups and using slice color, programs can make full use of L2 cache to improve performance. At the same time, we rethink the vertical allocation of L2 and L3 cache in page color. By allocating more L2 cache to L2 cache-sensitive programs and more L3 cache to L3 cache-sensitive programs concurrently, the system performance is improved, and some allocation strategies are proposed. Finally, we make full use of slice information and propose partial conflict color. Experiments show that partial conflict color can make system performance better when the colors are insufficient.

REFERENCES

[1] Kim, Taesoo, M. Peinado, and G. Mainar-Ruiz. "StealthMem: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud." Usenix Conference on Security Symposium 2013.

[2] Hund, Ralf , C. Willems , and T. Holz . "Practical Timing Side Channel Attacks against Kernel Space ASLR." IEEE Symposium on Security & Privacy 363.9418(2013):191-205.

[3] Yarom, Yuval, Qian Ge, Fangfei Liu, Ruby B. Lee and Gernot Heiser. "Mapping the Intel Last-Level Cache." IACR Cryptology ePrint Archive 2015 (2015): 905.

[4] Maurice, Clmentine, Scouarnec, N. L. , Neumann, C. , Heen, O. , & Francillon, Aurlien. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. " International Workshop on Recent Advances in Intrusion Detection Springer International Publishing, 2015.

[5] Wei, Zhipeng , Z. Cui , and M. Chen . "Cracking Intel Sandy Bridge's Cache Hash Function." Computer Science (2015).

[6] Lin, J. , Lu, Q. , Ding, X. , Zhang, Z. , Zhang, X. , & Aurlien Francillon. "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems." 2008 IEEE 14th International Symposium on High Performance Computer Architecture IEEE, 2008.

[7] Zhao, L. , Iyer, R. , Upton, M. , & Newell, D. "Towards hybrid last level caches for chip-multiprocessors." ACM SIGARCH Computer Architecture News 36.2(2008):56.

[8] Soares, Livio , D. K. Tam , and M. Stumm . "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer." 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), November 8-12, 2008, Lake Como, Italy ACM, 2008.

[9] Ye, Y. , West, R. , Cheng, Z. , & Li, Y. "COLORIS:a dynamic cache partitioning system using page coloring." International Conference on Parallel Architecture & Compilation Techniques IEEE, 2014.

[10] Wang, X. , Chen, S. , Setter, J. , & Martinez, J. F. "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support." 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA) IEEE, 2017.

[11] 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.

[12] El-Sayed, N. , Mukkara, A. , Tsai, P. A. , Kasture, H. , Sanchez, D. "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores." 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA) IEEE, 2018.

[13] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, Jr., and Dejan Kosti. 2019. "Make the Most out of Last Level Cache in Intel Processors". In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). ACM, New York, NY, USA, Article 8, 17 pages. DOI: https://doi.org/10.1145/3302424.3303977

[14] Huang, Y. , Chen, L. , Cui, Z. , Ruan, Y. , Bao, Y. , Chen, M. , et al. "HMTT: A Hybrid Hardware/Software Tracing System for Bridging the DRAM Access Trace's Semantic Gap." Acm Transactions on Architecture Code Optimization 11.1(2014):1-25.

[15] J. Chang and G. S. Sohi. "Cooperative cache partitioning for chip multiprocessors." In Proc. ICS07, 2007.

[16] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource." In Proc. PACT06, pages 1322, 2006.

[17] K. J. Nesbit, J. Laudon, and J. E. Smith. "Virtual private caches." In Proc. ISCA 07, 2007.

[18] M. K. Qureshi and Y. N. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." In Proc. MICRO06, pages 423432, 2006.

[19] N. Rafique, W.-T. Lim, and M. Thottethodi. "Architectural support for operating system-driven CMP cache management." In Proc. PACT06, pages 212, 2006.

[20] Xu, M., Phan, L. T. X., Choi, H. Y., & Lee, I. "vCAT: Dynamic Cache Management Using CAT Virtualization." Real-time Embedded Technology Applications Symposium 2017.

[21] Kim, D. , Kim, H. , Kim, N. S. , Huh, J. "vCache: Architectural support for transparent and isolated virtual LLCs in virtualized environments." IEEE/ACM International Symposium on Microarchitecture ACM, 2015.

Authorized licensed use limited to: University of Chinese Academy of SciencesCAS. Downloaded on June 20,2022 at 02:30:58 UTC from IEEE Xplore. Restrictions apply.