J Supercomput (2014) 69:547–560
DOI 10.1007/s11227-014-1092-2

Lightweight dynamic partitioning for last-level cache of multicore processor on real system

Ludan Zhang · Yi Liu · Rui Wang · Depei Qian

Published online: 20 January 2014
© Springer Science+Business Media New York 2014

Abstract With rapid development of multi/many-core processors, contention in shared cache becomes more and more serious that restricts performance improvement of parallel programs. Recent researches have employed page coloring mechanism to realize cache partitioning on real system and to reduce contentions in shared cache. However, page coloring-based cache partitioning has some side effects, one is page coloring restricts memory space that an application can allocate, from which may lead to memory pressure, another is changing cache partition dynamically needs massive page copying which will incur large overhead. To make page coloring-based cache partition more practical, this paper proposes a malloc allocator-based dynamic cache partitioning mechanism with page coloring. Memory allocated by our malloc allocator can be dynamically partitioned among different applications according to partitioning policy. Only coloring the dynamically allocated pages can remit memory pressure and reduce page copying overhead led by re-coloring compared to all-page coloring. To further alleviate the overhead, we introduce minimum distance page copying strategy and lazy flush strategy. We conduct experiments on real system to evaluate these strategies and results show that they work well for reducing cache misses and re-coloring overhead.

Keywords Multicore · Cache partition · Dynamic re-coloring · Page copying · Malloc allocator

1 Introduction

Multicore processors have become prevalent and hit the market at all fronts. The trend of integrating many-cores on a single chip can make performance more sensitive to

---

L. Zhang · Y. Liu (✉) · R. Wang · D. Qian
Department of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: yi.liu@jsi.buaa.edu.cn

contention for shared resources on multicore processors, especially at shared cache level. Shared cache contention will significantly exacerbate the existing memory wall and restrict the performance benefit of multicore processors.

Cache partitioning is a solution to reduce cache contention among co-running threads/processes. However, without particular hardware support, most research works conducted cache partitioning by simulation which is lack of accuracy. While in most modern architectures, caches are physically indexed and way associative, recent efforts have implemented software cache partitioning with page coloring [1,2], making it possible to partition cache on most commercial computers. However, page coloring-based cache partitioning has some inevitable side effects on performance of systems [3]. One is that page coloring restricts memory space that an application can allocate. An application may suffer from memory pressure when pages of its assigned color ran out, while pages of colors assigned to other applications are abundant. Another serious problem is the overhead caused by dynamic re-coloring. When dynamically changing cache partition, cache space assigned to the application is adjusted through changing its color-set and copying data from old pages to new pages whose color is within the new color set. However, frequent re-coloring may cause more overhead than the benefit of cache partitioning.

To make page coloring-based cache partitioning more practical, we only color dynamically allocated pages. As accessing data in pages allocated at run time is one of the main cause of shared cache contention, only coloring dynamically allocated pages can reduce cache contentions, while at the same time, reduce the cost of copying pages for dynamic re-coloring compared to all-page coloring.

Thus, this paper implements dynamic cache partitioning combined with a cache and application-aware malloc allocator. The allocator allocates pages according to the color-set assigned to the application. Our partition policy assures that different applications allocate pages in different color-set, hence their accessed region in shared cache is partitioned. Cache partition can be adaptively adjusted by changing the color of pages allocated by the malloc allocator. To alleviate the overhead of dynamic re-coloring, this paper proposes a minimum distance page copying strategy to reduce the number of page copying and introduces page copying delay strategy to avoid unnecessary page copying.

The contributions of this paper include: (1) in order to reduce the side effects of page coloring, we implement cache partition through a cache and application-aware memory allocator which only color dynamically allocated pages; (2) our minimum distance page copying strategy and lazy flush strategy can reduce dynamic re-coloring overhead further, making page coloring-based cache partitioning more practical. (3) We propose a type recognition-based cache partitioning policy and introduce the type recognition approach in this paper. (4) Our approach needs no changes on either source code of the application or the OS kernel.

Experiments show that our approach can effectively improve performance of co-running applications through cache partitioning for as high as 14.28 %; We can draw the conclusion that only coloring the dynamically allocated pages can reach the purpose of reducing cache contention miss. Overhead caused by dynamic re-coloring can be reduced significantly through our minimum distance page copying strategy(more than 55 % on average) when partitioning frequency is high. The minimum distance page

copying policy is more beneficial to application with larger dataset and shorter data reuse distance.

The rest of this paper is organized as follows: Section 2 introduces related work. Section 3 details the malloc allocator-based dynamic cache partitioning mechanism and the strategy to reduce re-coloring overhead. These are evaluated in Sect. 4 and we conclude the paper in Sect. 5.

2 Related work

2.1 Page coloring

In physically indexed and set-associative caches, physical addresses of data are used to map data into cache sets. The bits for hashing data into cache are represented by the color of the page.

Page coloring can be used to control the mapping from virtual memory to processor cache. By controlling color-set assigned to an application, we can control the exact cache region the application can access.

Page coloring technology was first used to ensure the stability of the program's performance, [4–6] used page coloring to enhance the performance of a single program. In recent years, page coloring has been applied to cache partitioning [1,2].

What relates to page coloring is dynamic re-coloring. Dynamic re-coloring can change the data placement of the process, making cache partition adjusted at runtime. However, this operation involves massive page copying which incurs high overhead.

2.2 Software cache partitioning

Software cache partitioning uses page coloring technology to reduce contentions in shared cache. Zhang et al. [1] noted that although the overhead caused by dynamic re-coloring cannot be ignored, software cache partitioning makes it possible to study the cache partitioning strategy on actual machine, which is more accurate than simulation. But they excluded the re-coloring overhead for the purpose of evaluating hardware-based scheme. ULCC [7] is a software library, programmer can use functions provided by ULCC to manage and optimize the use of shared cache in multicore. However using ULCC to manage shared cache requires modifications to source code of the application which set high requirements for the programmer and the management is static. Soft-OLP [8] is an object-level cache partitioning tool, which uses binary instrumentation techniques to calculate the reuse distance of the object which conduct their partitioning policy. While binary instrumentation makes the program slowdown significantly, Soft-OLP still does not have generality. Ccontrol [9] provides an open-source software and one can use it to restrict the cache region that an application can access, but it is a static cache partition software without any partitioning strategy [3] is the closest work related to our approach, to alleviate the coloring-induced adverse effects in practice, they proposed a hot-page coloring approach only enforcing coloring on most frequent visited pages. They determine the most frequent accessed pages by traversing the page table. However, this requires modifications to the OS kernel and traversing page table is also a big time consumer. They focused on how to determine the heat of each page and reduce

the overhead of traversing page table. Our work also employed partial page coloring. While [3] only color K hottest color, they need to compute and record access frequency of each page, which incurred additional overhead. Their approach may encounter error prediction sometimes which will make page re-coloring meaningless. To avoid such additional overhead, we only color the dynamically allocated pages which are also frequently accessed. We focused on how to make re-coloring overhead minimal.

3 Malloc allocator-based dynamic cache partitioning mechanism

Our malloc allocator-based dynamic cache partitioning mechanism includes two stages. First, when an application asks for pages at runtime, our malloc allocator will allocate pages according to the color-set assigned to the application. Our partition policy assures that different applications allocate pages in different color-set, hence their accessed region in shared cache is partitioned. When the co-running applications change, the color-set of the running application will be adjusted adaptively, then copying data from old pages to new pages whose color is in the new color-set. While physical pages that containing the data in heap memory change, the cache region that those data mapped to is also changed.

The overall design of our mechanism include four components: loadable module, dynamic malloc allocator, page copying component and policy maker. We manage pages according to their colors with a dynamically loadable module, which provides interface to other components to manage and color pages of a given process. An environment variable, LD_PRELOAD, is used to substitute application's dynamic link library so that its requests for memory at runtime is satisfied by our malloc allocator. Page copying component is activated when changing cache partition and it reduces overhead caused by page re-coloring. The policy maker makes partitioning decision adaptive and provides information needed by malloc allocator and page copying component.

We will introduce two strategy here, one is type recognition-based partition policy which decides color-set assigned to each co-running application; the other is the strategy to reduce re-coloring overhead of changing cache partition. This section details the type recognition-based partition policy and re-coloring overhead reducing strategy.

3.1 Type recognition-based partition policy

3.1.1 Partition policy

Our adaptive cache partition policy adjusts partition dynamically according to the type and miss-ratio curve of co-running applications. The applications are classified into four types according to their cache access patterns. Applications of type A are very sensitive to the last-level cache size. Applications of type B are little sensitive to last-level cache size. Applications of type C and D are last-level cache size inelastic, while cache miss of type C remains high and that of type D keeps low. Applications from type A to type D have priorities from the highest to the lowest. We categorize applications into four types because our cache partition policy favors cache size demand

for applications with higher priority. We define cache size demand for an application as the minimum cache size to guarantee its performance. Our policy calculates the cache size demand according to miss-ratio curve. It is the point from where the miss-ratio changes with cache size gently, before the point miss-ratio drops with cache size becoming larger. To improve overall system performance when two applications of different types co-running sharing last-level cache, the cache size demand of application with higher priority is satisfied first and assign more cache size to it. When applications with the same type run simultaneously, cache size are increased for each application according to the proportion of their cache size demand.

A type recognition approach is used to identify the type of an application, which is discussed in the following.

3.1.2 Type recognition approach

(a) Pattern normalization: to identify the type of an application, we measure last-level cache misses under various partition sizes and generate its miss-ratio curve. To normalize the sampled data, a coarse granularity coordinate is introduced that treats every 16 colors as one unit on the X axis, and every five miss-ratio as one unit on the Y axis. The normalized feature is extracted via the following process:

Step 1 First, calculate the coarse point in the coarse coordinate. As our system has 64 colors, we can reduce point on miss-ratio curve to four points—the value of x coordinate is from 1 to 4. The value of y coordinate is the average value the miss-ratio curve passed by in the interval which is represented by coarse x coordinate.

Step 2 Second, calculate the coarse slope value between each consecutive coarse point. The slope list has three value, because that there are four coarse points. Coarse slope value is defined by Eq. (1), if two consecutive points in the coarse coordinate are (x_1,y_1) and (x_2,y_2), respectively.

$$\text{slope} = \frac{y_2 - y_1}{x_2 - x_1} \tag{1}$$

Step 3 The normalized pattern is defined as a bit string with 3 bit according to slope value list. If the slope value is non-zero, then its corresponding bit is 1, otherwise is 0. Non-zero means miss-ratio decreases sharply with cache size becoming larger and zero means miss-ratio is not sensitive to the cache size in the interval.

As Fig. 1 shows, the benchmark sphinx has four coarse points (1,13), (2,12), (3,10) and (4,7). The corresponding coarse slope value between each two consecutive points is -1, -2, -3. Thus, the pattern of sphinx is "111". We normalize the pattern of the application with bit string, making it possible to recognize the pattern of an application with little normalized point and those points contain enough information.

(b) Pattern recognition: if the normalized pattern has three bits as "1" , it is recognized as type A. Similarly, an application is recognized as type B if its pattern contains

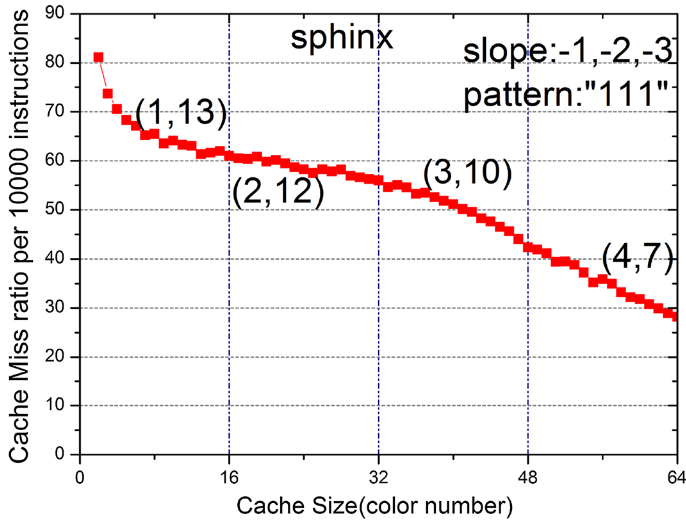

Fig. 1 Normalized pattern of sphinx

one or two non-zero bits. For ”000” pattern, if the coarse point on *Y* axis is bigger than 1, then the type is C, otherwise the type is D. Through the approach, sphinx is recognized as type A.

3.2 Re-coloring overhead reducing strategy

Our mechanism changes cache partition adaptively by changing the color-set of dynamically allocated pages. On one hand, it is implemented by copying data from old pages to new pages in the new color-set. On the other hand, malloc allocator will allocate pages according to the new color-set from then on. In order to reduce the overhead of page copying, it is necessary to minimize the number of page copying and postpone the copying operation.

3.2.1 Minimum distance page copying policy

After new color-set is calculated, we then assign color in the new color-set to each page. Continuous allocated pages should have different colors, the reason is that according to the locality principle, spatially continuous pages are often accessed temporal continuously. If continuous accessed pages have the same color, it is likely to make that cache region become a hot spot. Also, colors cannot be assigned arbitrarily to keep color balance. Color balance means pages of an application should scatter cross colors assigned to it and the page number of each color should be almost the same. First, we calculate page number of each color in target color-set in a balanced way (almost equal). In order to reduce the number of page copying, we maintain a current_map array which records the current color of each page, we will calculate the target_map array which stores target color of each page after re-coloring. Among the coloring decision with

page number of each target color determined, the distance between target_map array and current_map array is the minimum. The distance between two arrays refers to the number of different items between them. The pseudo-code is shown in Algorithm 1.

```
Input:  current_map array , N : number of pages , CN : target color number
Output:  target_map array
 1: //count[1...CN] records page number of each target color after remapping
 2: count[1...CN]=N/CN
 3: count[1...N%CN]++
 4: for i = 1 to N do
 5:    c = current_map[i] //c is current color of page i
 6:    //if the assignable number of color c is greater than 0
 7:    if count[c] >0 then
 8:       //maintain the current color of page i as c
 9:       target_map[i]=c
10:       //update the assignable page number of color c
11:       count[c]- -
12:    else
13:       //the assignable number of color c is 0
14:       choose a new color nc who has enough assignable page in round-robin way
15:       target_map[i]=nc //change color of page i to nc
16:       // update the assignable page number of color nc
17:       count[nc]- -
18:    end if
19: end for
```

Algorithm 1: Minimum distance page copying

After the target_map[i] is calculated, if it is not equal to current_map[i], the color of page i needs to be changed. As the distance between target_map and current_map is the shortest, the number of pages whose color needs to be modified is the minimal and the number of page copying is also minimized.

3.2.2 Page copying delay strategy

We implement lazy page copying (proposed by Zhang et al. [1]) which delays page copying to the time of first access. We have strengthened lazy page copying with lazy flush policy. Delaying page copying to the time of first access, on one hand, can prevent memory allocation from becoming performance bottleneck. On the other hand, before the page being accessed, its color may be modified again, delaying copying can avoid such meaningless page copying. To modify the color of a page from A to B, we first set the target color of the page as B, then clear the the corresponding entry of the page in page table entry. After that, any subsequent access to that page will trigger page fault error, at that time our page fault handler will be invoked and copy data from old page to new page with color B and free the original page.

The following describes our lazy flush policy.

After the entry of the page in page table is cleared, the original page is still cached and the old value of page table entry is also cached in TLB. So we need to refresh the data cache and invalidate TLB. Since the refresh operation often leads to compulsory

misses, it should be executed as less as possible. To satisfy this requirement, lazy flush strategy is proposed: traversing the target_map and current_map array after the target_map array being calculated, if the value of current_map and target_map is different from the current page, then clear the entry of the page in page table. Finally, flushing cache and TLB until all pages are processed.

3.2.3 Analysis of overhead

In principle, the overhead of re-coloring is composed of two parts: the first part is overhead of copying pages from old colors to new ones; and the second part is overhead of re-coloring algorithm itself. Compared to traditional solutions, our approach only coloring dynamically allocated pages instead of all pages, hence the number of page copying can be reduced significantly. As for the overhead of re-coloring algorithm, it can be concluded from pseudo-code shown in Algorithm 1 that the complexity of our minimum distance page copying policy is O(N), where N is the number of pages.

4 Evaluation

Our approach is evaluated on a dual-core Intel Core 2 platform, of which each core has a 32KB, 8-way associative private cache, and two cores share a single 6MB, 24-way set-associative L2 cache. Page size is 4KB and physical pages can be divided into 6MB/(4K $\times$ 24) = 64 colors. We use Ubuntu 10.04 with linux kernel 2.6.32. Execution performance data are collected using perf [10]. 12 benchmark applications from SPEC CPU2006 [11] are used in our experiment, which can be categorized to four types according to our type recognition approach, shown in Table 1.

The miss-ratio curve of the 12 benchmarks is shown in Fig. 2.

4.1 Evaluation of performance

To evaluate performance improvement of our adaptive type recognition-based cache partitioning policy, we select 10 out of the 12 benchmarks and divide them into two groups as:

group 1 = {mcf, GemsFDTD, libquantum,lbm}

Table 1 Category of benchmarks

Type	Benchmarks
A	Bzip2, mcf, sphinx
B	H264ref, GemsFDTD, libquantum
C	zeusmp, bwaves, lbm, sjeng
D	gromacs, namd

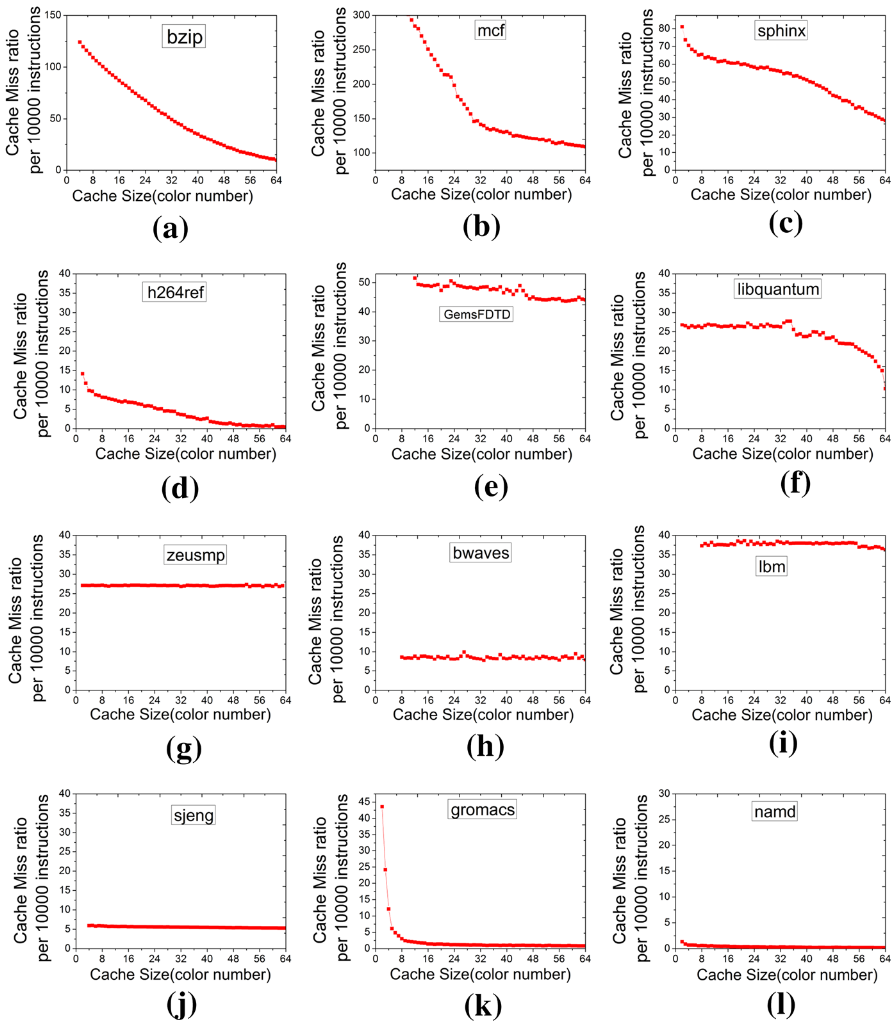

Fig. 2 Miss-ratio curve of benchmarks

group 2 = {bzip2, mcf, sphinx, h264ref, GemsFDTD, libquantum, zeusmp, bwaves, lbm, gromacs}

Each time we select a benchmark from group1 and the other from group2, co-locate them on neighboring two cores sharing L2 cache. We compare system performance under two policies:

1. In default sharing, applications freely compete for the shared cache space.
2. In type recognition-based partitioning policy, cache partitioning is conducted by the type and miss-ratio curve of co-running applications. When an application's co-runner changes, the partition size will be re-arranged adaptively.

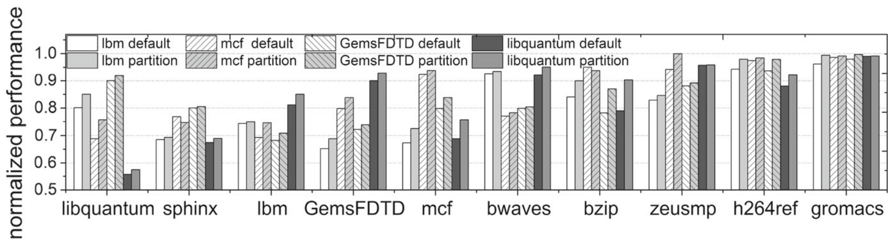

Fig. 3 Normalized performance of co-running applications

Supposing the execution time of each two application is t_{s1} and t_{s2} when they run alone and utilizing the whole cache. When they run together sharing the L2 cache, their corresponding execution time is t_{p1} and t_{p2}. Our performance metric is defined by Eq. (2):

$$p = \frac{1}{2}\left(\frac{t_{s1}}{t_{p1}} + \frac{t_{s2}}{t_{p2}}\right) \tag{2}$$

Then the ideal performance is 1.0 if two applications co-run without any interference.

Figure 3 shows normalized performance of the system when application on the *X* axis co-runs with lbm, mcf, GemsFDTD and libquantum under two policies. The results demonstrate that our partition policy achieves performance improvement as high as 14.28 and 3.82 % on average. The average value is not high because that not all of applications benefit much from cache partitioning due to their work-set and access pattern. For example, co-running with libquantum, lbm, GemsFDTD and mcf, sphinx causes high performance degradation to the system compared to running alone, due to their heavy demand on shared cache which is consistent with the conclusion of [12]. H264ref and gromacs have little impact on system performance. Hence it benefits little from cache partitioning when co-running with h264ref and gromacs. When lbm,libquantum,GemsFDTD and mcf co-runs with bzip, our policy can achieve 10.84 % performance improvement on average and only 4.26 % on average when they run simultaneously with h264ref. The performance of two sets—mcf vs. Bzip and mcf vs. sphinx—declined compared to default sharing, it is mainly because that mcf, bzip and sphinx are applications of type A, which is sensitive to the cache size. When two applications of type A co-run, cache partitioning cannot meets both requirements in cache size, due to their high demand in cache size. Although cache partitioning can reduce contention miss between two applications, it will increase capacity miss at the same time. The performance degradation due to a substantial increase in capacity miss negates the benefit gained by the reduction of contention miss. So partition policy requires trade-off between the capacity miss and contention miss caused by cache partitioning.

4.2 Evaluation of overhead

To evaluate re-coloring overhead, which depends on frequency of the cache partitioning operation, we execute partitioning operation in different time intervals from 600 to 1,000 ms, and run only one application each time to eliminate interferences from other applications. We change the cache size from 44 to 64 colors for each time interval. Cache size for 64 colors corresponds to the whole cache area. Size corresponding to 44 colors is larger than most application's cache size demand except that of type A on our platform, so the interference of capacity miss can be reduced. We compare the cost under two policies:

1. In default page copying policy, we re-assign target colors to pages in a round-robin way, distributing pages to all assigned colors in a balanced way just as other research did.
2. In minimum distance page copying scheme, we re-assign target colors to pages according to the calculated target_map array and at the same time keep the color balanced.

The evaluation uses four benchmarks:lbm(type C, memory intensive), sjeng(type C, memory intensive), sphinx(type A,compute intensive) and libquantum(type B, compute intensive).

The re-coloring cost metric is defined as re-coloring number per page, that is, total number of page copying divided by page number of the application. Figure 4 shows the cost of each application. Our minimum distance policy can reduce re-coloring cost by more than 55 % on average (71.27 % for lbm, 70.63 % for sjeng, 36.95 % for sphinx and 40.53 % for libquantum) compared to the default policy. Re-coloring cost decreases as the partitioning time interval increases gradually. The figure also shows that re-coloring cost of lbm and sjeng is high, the reason is that work-set of these two applications is large and data reuse distance is short, while the cost of sphinx and libquantum is low, because that their data reuse distance is longer and work-set is smaller relatively.

Figure 5 shows normalized execution time of each application under two different page copying policies, the execution time is normalized to that of running alone without interference from other applications and cache partitioning. The figure shows that execution time decreases when the partitioning time interval becomes longer. The minimum distance page copying policy achieves better performance than the default policy. lbm and sjeng are memory intensive and have large dataset, hence frequent changes of their cache partition trigger massive page copying, making their execution time increased significantly. The result is consistent with Fig. 4. However, adjusting cache partition frequently makes execution time increased significantly for sphinx because that it is type A and sensitive to cache size. Execution time of libquantum is not affected that much because its re-coloring number per page is low and it is not that sensitive to cache size. Our policy improved their performance by 16.49 % for lbm, 12.28 % for sjeng, 1.43 % for sphinx and 0.38 % for libquantum compared to the default one. The minimum page copying policy contributes little performance improvement for sphinx and libquantum, the reason is that their dataset is small and they are compute-intensive. We can draw the conclusion that re-coloring cost is related with the reuse

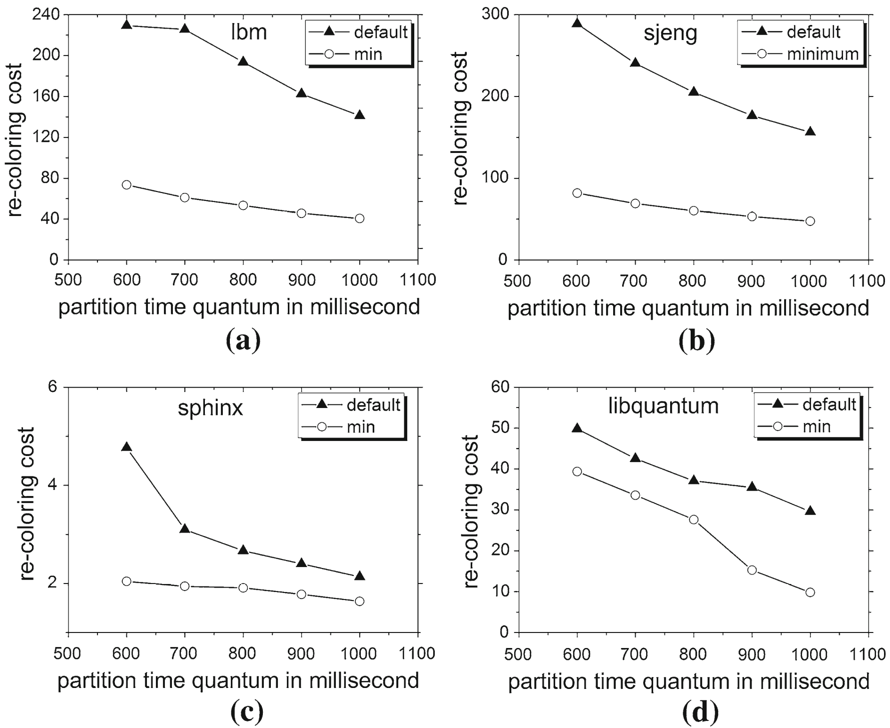

Fig. 4 Re-coloring cost under two different page copying policies

distance, size of dataset, memory access pattern of the application and the frequency of cache partitioning operation. The influence of reuse distance is due to delaying page copying to the time it is accessed. Thus, the minimum distance page copying policy is more beneficial to applications with larger dataset and shorter data reuse distance.

5 Conclusion and future work

In order to reduce shared cache competitions in multicore processors and make page coloring-based cache partition more practical, this paper presents a malloc allocator-based cache partitioning mechanism with dynamic page coloring, in which memory allocated by our malloc allocator can be dynamically partitioned among different applications according to partitioning policy. To further reduce overhead of page copying led by re-coloring, a minimum distance page copying strategy and lazy flush strategy are proposed in this paper.

Our approach is implemented on real system. Experiment results show that our adaptive cache partition policy can improve performance of co-running applications, and to achieve better performance, trade-off between contention miss and capacity miss is required for partition policy. The cost of re-coloring can be alleviated significantly

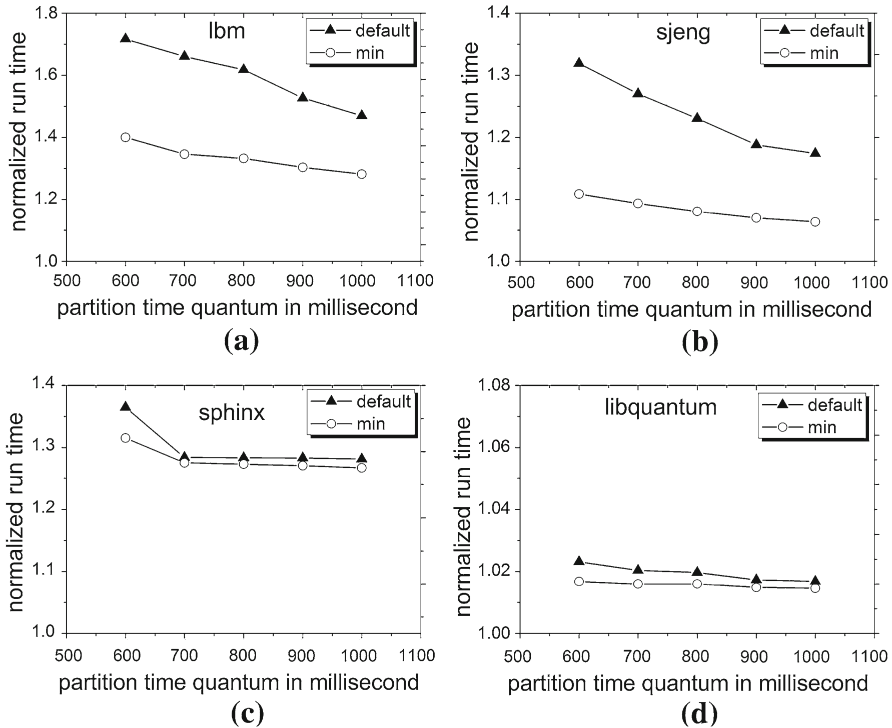

Fig. 5 Normalized execution time of applications under two page copying policies

by our minimum distance page copying policy, which is more beneficial to applications with larger dataset and shorter data reuse distance.

In future work we will focus on two aspects. First, how to implement malloc allocator-based cache partitioning mechanism on emerging many-core processors, in which last-level shared cache tends to be distributed into processing cores, namely tiles, making access delay different for different region of the cache; second, how to support real applications such as [13] rather than benchmarks generally consisted of small programs with small dataset.

Acknowledgments We thank the anonymous reviewers for their insightful comments, which greatly improved the quality of this manuscript. This work is supported by National Science Foundation of China under Grant No. 61073011 and 61133004, and National High-Tech Program of China (863 program) under Grant No. 2012AA01A302.

References

1. Lin J, Lu Q, Zhang X et al (2008) Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems. In: Proceedings of the 14th international symposium on high performance computer architecture (HPCA-14), Salt Lake City
2. Soares L, Tam D, Stumm M (2008) Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In 41th international symposium on microarchitecture

3. Zhang X, Dwarkadas S, Shen K (2009) Towards practical page coloring-based multicore Cache management. In: Proceedings of the 4th ACM European conference on computer systems (EuroSys'09), pp 89–102
4. Taylor G, Davies P, Farmwald M (1990) The TLB sliceCa low-cost high-speed address translation mechanism. In: Proceedings of the ISCA'90, pp 355–363
5. Kessler RE, Hill MD (1992) Page placement algorithms for large real-indexed caches. ACM Trans Comput Syst 10(4):338–359
6. Bugnion E, Anderson J, Mowry T et al (1996) Compiler-directed page coloring for multiprocessors. ACM SIGPLAN Not 31(9):244–255
7. Ding X, Wang K, Zhang X (2011) ULCC: a user-level facility for optimizing shared cache performance on multicores. In: Proceedings of 16th ACM SIGPLAN annual symposium on principles and practice of parallel programming (PPoPP 2011), 12–16 Feb 2011
8. Lu Q, Lin J, Zhang X et al (2009) Soft-olp: improving hardware cache performance through software-controlled object-level partitioning. In: Proceedings of the 18th international conference on parallel architectures and compilation techniques (PACT), pp 246–257
9. Perarnau S, Tchiboukdjian M, Huard G (2011) Controlling cache utilization of hpc applications. ACM. In: Proceedings of the international conference on supercomputing, pp 295–304
10. perf. http://perf.wiki.kernel.org/.2011
11. SPEC CPU2006. http://www.spec.org/cpu2006.2006
12. Tang L, Mars J, Soffa ML (2011) Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In: Proceedings of the 1st international workshop on adaptive self-tuning computing systems for the Exaflop Era, San Jose, June 2011
13. Zhu X, Li K, Salah A (2013) A data parallel strategy for aligning multiple biological sequences on multi-core computers. Comput Biol Med 43(4):350–361