ICIC Express Letters
Part B: Applications
Volume **7**, Number **10**, October **2016**

 ISSN 2185-2766
pp. **2149–2154**

EFFICIENT PAGE COLORING FOR MULTI-THREADED PROGRAMS

Xiuhong Li and Gulila · Altenbek

Information Science and Engineering College
Xinjiang University
No. 666, Shengli Road, Urumq 830046, P. R. China
{ xjulxh; gla }@xju.edu.cn

Received March 2016; accepted June 2016

Abstract. *Cache is introduced in modern processors to accelerate data accessing. However, it happens commonly that good locality data in cache are often evicted by poor locality data, resulting into poor cache utility and degraded performance. This paper proposes that the lock in multi-threaded programs naturally reveals the data locality information which can be leveraged to achieve better page coloring and thus better cache utility. Experimental results show that our work can achieve an average speedup of 1.25X with only ignorable overhead (less than 5%), showing great potential to accelerate multi-processing.*
Keywords: Cache utility, Page coloring, Data locality, Multi-threaded programs, Lock

1. **Introduction.** Modern processors all introduce cache to accelerate data accessing based on the principle of locality [1]. However, the data locality feature is only determined by the runtime behaviors of upper applications and varies hugely in different applications [2]. A common situation happens that some cache lines with high locality may be evicted by others with low locality, resulting into poor cache utility and degraded performance [3]. Better cache management strategy is thus desirable to improve cache utility.

Without any upper level (upper application) information, a common way to achieve better cache utility is to try to assign neighboring pages to different cache sets, hoping they have good locality and thus they will not conflict with each other in cache (we call this method basic page coloring or open mapping [3], see details in Section 2). In order to achieve better cache utility, there have been a lot of studies trying to get more information from upper applications to guide cache management, which include off-line analysis method [4], new-monitoring-hardware-based method [5], and memory-allocator-based method [3]. These previous studies all fall short on some aspects. Off-line method and new-hardware-method are not practical in most cases. Memory-allocator-based method introduces considerable overhead when doing monitoring and analysis at on-line execution [3] and some studies changed the memory allocation interface [6]. Overall, a high efficient and transparent on-line method is needed to achieve better cache management.

This paper aims to achieve better cache management (better page coloring) for multi-threaded programs. We conduct our work based on a key insight: multi-threaded programs offer very natural information which reveals its memory access pattern. When a thread tries to hold a lock to access some shared data, it reveals that the programmers assume this shared data may be accessed soon by other threads. Thus this shared data is supposed to have good locality. We will show in this paper that this information is very effective and is easy to get at runtime, introducing only ignorable overhead. Based on this key insight, we have built our tool which contains a kernel patch and a runtime library that offers lock operation interface and memory allocation interface. The lock operation interface is the same with the *pthread* library to catch the lock information. The memory allocation interface is the same with the *libc* library to achieve memory access monitoring.

The kernel patch manages the virtual-physical mapping to achieve better cache management (page coloring) based on the information we get from the upper multi-threaded applications. Any application can run with our tool without any modification. Experiments show that our work achieves an average speedup of 1.25X. The on-line analysis only introduces less than 5% overhead.

The rest of this paper is organized as follows. We introduce some background information about cache management and page coloring in Section 2. The design and implementation details of our method are in Section 3. We conduct experiments in Section 4 and conclude in Section 5.

2. **Background Information about Page Coloring.** Modern processor organizes cache lines into several cache sets and uses physical address to manage cache and physical memory. As shown in Figure 1, different addresses in physical memory are assigned to different cache sets in cache. Based on the locality principle, all the cache lines in two neighboring physical pages are assigned to different cache sets to avoid conflict in cache and for this we say the two neighboring physical pages have different colors. For example, in Figure 1 there are two colors (red and blue). All the red physical pages share the same group of cache set and all the blue physical pages share the other same group of cache set. Thus any two cache lines in different colored physical pages would never conflict with each other in cache.

The cache and physical memory management logic introduced above is set fixed in hardware and cannot be modified by software. However, software runs with virtual memory address instead of physical memory address. It is the operating system's task to manage the page table to map virtual pages to physical pages. Here we call this mapping process

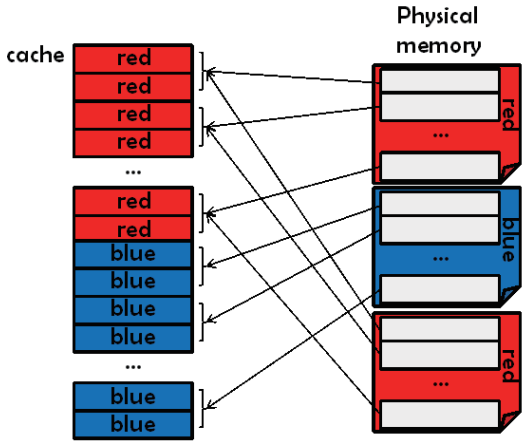

FIGURE 1. Cache management and page coloring

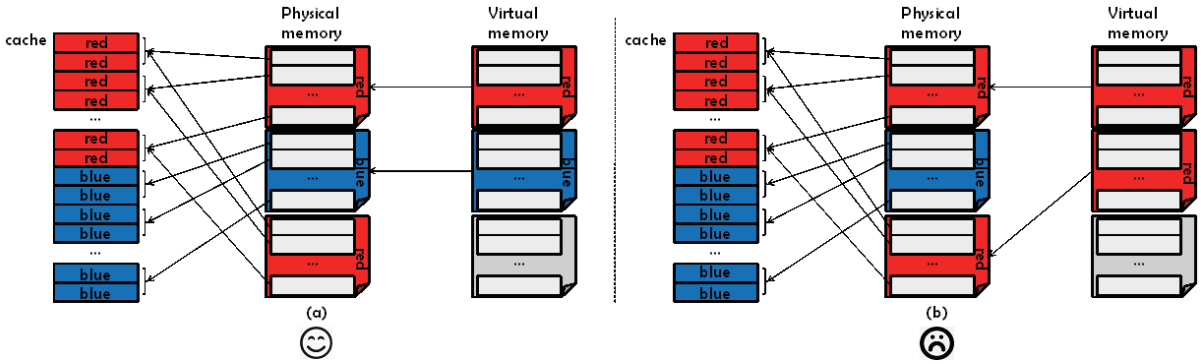

FIGURE 2. Page coloring with virtual-physical mapping

coloring. For example, in Figure 2(a), the second virtual page can be mapped to the second physical page and thus it is colored *blue*. Also in Figure 2(b), the second virtual page can be mapped to the third physical page and thus it is colored *red*. Operating system determines the color of virtual pages and thus determines which group of cache set the virtual page uses. Thus if we find the two neighboring pages have good data locality, it is better to give the two neighboring virtual pages different colors as shown in Figure 2(a). On the contrary, Figure 2(b) shows a bad case we want to avoid for good locality data.

3. **Efficient Page Coloring for Multi-Threaded Programs.** As discussed in Section 2, the key problem to achieve better cache management is to try to discover the data locality information in upper applications and use the information to guide page mapping in operating system. Our method for this includes a kernel patch which manages the virtual-physical mapping and a runtime library which discovers the data locality information of upper applications.

3.1. **Locks in multi-threaded programs.** Most multi-threaded programs rely on lock to synchronize accesses to shared data. The implicit information of lock is that if a thread holds a lock to access any shared data, this part of data is likely to be accessed soon by other threads. This is very natural information that the lock reveals. In most cases, the data in the inner-most lock sets is most intensively accessed and thus this part of data has good locality.

In order to catch the lock information, we offer a library that provides pthread_mutex_lock interface just the same with that in *pthread* lib. Thus upper applications need no modification to run with our library. In our implementation of pthread_mutex_lock, we just do some record and analysis and then call the real pthread_mutex_lock.

```
a = malloc(...);
b = malloc(...);
c = malloc(...);
...
pthread_mutex_lock(&l);          i_am_in_lock++;
                                 If(i_am_in_lock >= threshold)
                                     protect_shared_data();
                                 Real_pthread_mutex_lock(&l);
//access shared data a,b,c
a++;
b = a;
...
c++;
pthread_mutex_unlock(&l);        If(i_am_in_lock >= threshold)
                                     unprotect_shared_data();
                                 i_am_in_lock--;
                                 Real_pthread_mutex_unlock(&l);
```

FIGURE 3. Implementation of lock

Figure 3 shows our implementation of lock in our library. Firstly the thread allocates three objects a, b, c and then requires a lock to access them. At the lock time (pthread_mutex_lock), we increase a variable *i_am_in_lock* to indicate that we are now in lock set. For nested lock set, this variable also records the number of nested layers. Then after holding the lock, the thread begins accessing the shared data. There are two situations when accessing the shared data.

1. If the virtual page which contains the shared data has just been allocated and has never been accessed before, then a page fault is triggered and the operating system should allocate a new physical page and construct the virtual-physical mapping. This situation is what we are mainly looking for to achieve efficient page coloring. The operating system will check whether we are in a lock set (check the variable *i_am_in_lock*). If so, the operating system will try to allocate different colored physical

page for all the page faults triggered in the lock set. Thus we give all the shared data different colors in a lock set and they would not conflict with each other in cache. We argue that it is a common situation in multi-threaded programs that the first time of accessing a new allocated virtual page happens in lock sets. This is because new allocated virtual pages are allocated through heap (malloc) and thus the objects in them are shared among threads. Accessing them requires holding locks. Achieving page coloring for this situation would only introduce ignorable overhead because this is the first time of accessing one virtual page and allocating a new physical page. We do not have to do remapping for the virtual page which has already been mapped to a physical page. Doing remapping is expensive (it requires allocating a new physical page, copying the content of the old physical page to the new page, and then modifying the page table to redirect the mapping).

2. The virtual pages that contain shared data have already been mapped to physical pages. Accessing them would not cause any page faults. In this situation, we hope that all the virtual pages accessed in this lock set have been given different colors thus they do not conflict with each other in cache. The virtual-physical mapping relationship for these virtual pages may be set up in different lock sets thus some of them may have the same color. Normally we just ignore this situation. However, under some intense situations, it is better that we double-check it. We regard the nested lock sets as intense situation. The more layers of the lock set nests, the more intensive accesses the shared data will serve. Thus as Figure 3 shows, we set a threshold to test the number of nested layers. If it reaches the threshold (currently we set it to be 2, in accordance to our experimental experience that nested layers larger than 2 often show intense situations), we will write-protect all the shared data. Accessing the shared data in the lock set will cause page fault so we have chance to check if all the virtual pages accessed in this lock set have different colors. If we find that two virtual pages have the same color, we will do remapping. As introduced before, doing remapping is expensive and thus we just do it when the shared data are serving intensive accesses. Moreover, we have found that in most multi-threaded programs nested lock sets are in loops, which are definitely intensive cases.

3.2. **Memory allocator.** We also provide dynamic memory allocation interface (malloc) in our library. When serving malloc, we just record all the virtual pages that are visible to programmers (the virtual pages that contain objects returned from malloc) and then call into the real malloc in *libc*. Thus when we want to write-protect all the shared virtual pages (do protect_shared_data), we can just use mprotect to achieve this. Accessing these virtual pages will cause page faults and thus by doing this we are able to check the colors of these virtual pages.

4. **Experimental Results.** In this paper we propose an efficient way to get the data locality feature in multi-threaded programs and to achieve better page coloring and better cache utility. In this section we will mainly show the benefit and overhead of our work. Also we will compare our work with the state-of-the-art work NightWatch [3], a memory-allocator-based page coloring tool.

We conduct our experiment on a platform with Intel processor E3 e1230 v3. It has 8 MB of last level cache and 256 colors. We test our work on multi-threaded benchmarks selected from SPLASH-2 [6] and PARSEC [7] benchmark suite with default input.

Figure 4 shows the experimental results. Firstly, Figure 4(a) shows the speedup of our work (lock-based) and previous state-of-the-art work (NightWatch) over the baseline (normal execution). We can see both our work (lock-based) and the previous work (NightWatch) achieve obvious speedup. Our work behaves much better than NightWatch for the benchmark *ocean, water, lu*, and *fft*. This is because these benchmarks only allocate

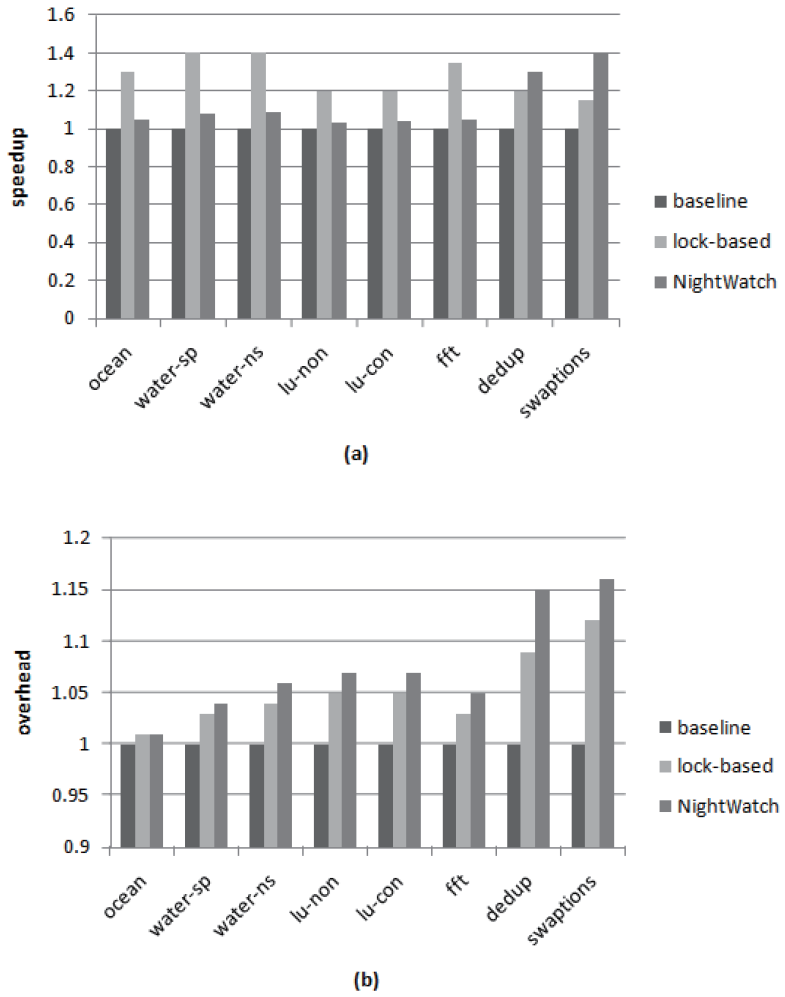

FIGURE 4. Experimental results

a small number of objects (less than 10) at the beginning of program and then make no memory allocations at runtime. NightWatch relies on memory allocation information to gain benefit and thus it behaves not so well. For the benchmark *dedup* and *swaptions*, they allocate a lot of objects and thus they give NightWatch abundant information. NightWatch behaves better than our work on these two benchmarks. Overall, our work relies on the lock information and achieves 1.25X speedup averagely.

To test the overhead, we just run our work and NightWatch with monitoring and analysis process but we do not do any page coloring based on that analysis. Figure 4(b) shows the overhead of our work (lock-based) and previous work (NightWatch) over baseline. We can see in all benchmarks our work introduces less overhead than NightWatch. This is because the information from lock is very natural and easy to get. Overall, our work introduces less than 5% overhead averagely.

Lastly, our work is for multi-threaded programs while NightWatch is more general. However, we want to point out that our work is compatible with NightWatch and can be combined with it to achieve further benefit.

5. **Conclusions.** This paper introduces an efficient page coloring mechanism which leverages the lock in multi-threaded programs to gain the data locality information. With the upper level data locality information, we could achieve better page coloring and better cache utility. Experiments show that our work could achieve an average speedup of

1.25X, introducing only less than 5% overhead. Future work includes adopting our work for transactional-interface-based multi-threaded programs.

REFERENCES

[1] M. E. Wolf and M. S. Lam, A data locality optimizing algorithm, *ACM Sigplan Notices*, vol.26, no.6, 1991.

[2] X. N. Ding, K. B. Wang and X. D. Zhang, ULCC: A user-level facility for optimizing shared cache performance on multicores, *ACM Sigplan Notices*, vol.46, no.8, 2011.

[3] R. T. Guo et al., NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems, *2015 USENIX Annual Technical Conference*, Santa Clara, CA, 2015.

[4] D. L. Schuff, M. Kulkarni and V. S. Pai, Accelerating multicore reuse distance analysis with sampling and parallelization, *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[5] D. K. Tam et al., RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations, *ACM Sigarch Computer Architecture News*, vol.37, no.1, 2009.

[6] C. Bienia, S. Kumar and K. Li, PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors, *IEEE International Symposium on Workload Characterization*, 2008.

[7] C. Bienia et al., The PARSEC benchmark suite: Characterization and architectural implications, *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.