J Supercomput (2016) 72:1126–1151
DOI 10.1007/s11227-016-1645-7

Lowering the volatility: a practical cache allocation prediction and stability-oriented co-runner scheduling algorithms

Fei Wang[1] · **Xiaofeng Gao**[1] · **Guihai Chen**[1]

Published online: 5 February 2016

Abstract The accurate and quantitative analysis of the cache behavior in a Chip Multi-Core (CMP) machine has long been a challenging work. So far there has been no practical way to predict the cache allocation, i.e., allocated cache size, of a running program. Lots of applications, especially those that have many interactions with the users, cache allocation should be estimated with high accuracy since its variation is closely related to the stability of system performance which is important to the efficient operation of servers and has a great influence on user experience. For these interests, this paper proposes an accurate prediction model for the allocation of the last level cache (LLC) of the co-runners. With a precise cache allocation predicted, we further implemented a performance-stability-oriented co-runner scheduling algorithm which aims to maximize the number of co-runners running in performance-stable state and minimize the performance variation of the unstable ones. We demonstrate that the proposed prediction algorithm exhibits a high accuracy with an average error of 5.7 %; and the co-runner scheduling algorithm can find the optimal solution under the specified target with a time complexity of $O(n)$.

Keywords Performance model · Probability · Cache allocation prediction · Scheduling

✉ Xiaofeng Gao
gao-xf@cs.sjtu.edu.cn

Fei Wang
bomber@sjtu.edu.cn; fay96816@gmail.com

Guihai Chen
gchen@cs.sjtu.edu.cn

[1] Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

1 Introduction

With the CMP servers getting popular, programs are now running simultaneously on different cores sharing the LLC. The sharing of LLC can lead to a high performance variation which has negative effects on many aspects. For program that sharing LLC with co-runners having large performance variations, its cache allocation will fluctuate and the contention of memory bandwidth is exacerbated which further degrades the overall performance. We emphasize that in consideration of the growing core frequency and cache size, program performance has a steady improvement while its variation is not. In fact, a varying service latency hurts the user experience and thus debase the user evaluation of the programs especially for the web applications which have direct interactions with end users, i.e., latency variation sensitive. During the scheduling of applications, a stable average service latency is preferable than an average latency with large variation even though it has higher average IPC.

Figure 1 shows a four-application scheduling problem with two online machine learning applications and two web server applications. The average IPC of the whole system (i.e., all four applications) and the average service latency variance of both web applications under two schedulers are shown in the figure. For performance-oriented scheduling algorithm, to reach highest IPC, each machine learning thread is co-run with a web server application to keep CPU as busy as possible. On the other hand, the other schedule mapping has an average IPC drop at 8.94 % but the service latency variance of web server applications decreased 62 % in average which is a significant improvement of user experience. Obviously for some user experience critical situations the tradeoff between overall performance and stability is reasonable and attractive. Therefore, finding a solution to achieve a stable performance(i.e., lower latency variance) is desired for servers running this kind of applications.

As we try to lower the performance variations under an LLC-sharing context, it is important to find a co-runner schedule mapping which yields small performance

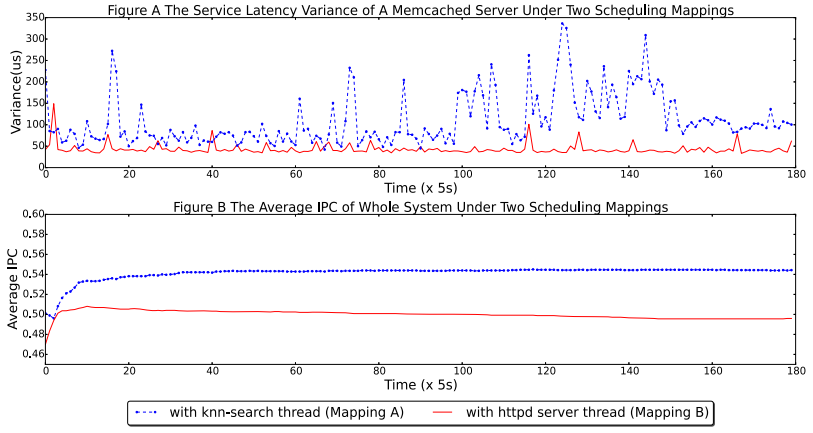


Fig. 1 The average performance and its service variation of a four-application schedule problem under different Schedules

variation, i.e., the co-runners being assigned under this schedule mapping suffer less performance variation than other mappings. To achieve this goal, an intensive research on the precise prediction of the cache allocation is required. Since for most programs, the contention of LLC allocation can drastically increase the data access latency and incur a rising performance variation which is hard to be predicted [1–3]. Generally speaking, the performance variation relies heavily on the cache allocation, i.e., a larger cache allocation means a higher hit ratio, and thus leads to a lower average data access latency and vice versa and thus a varying cache allocation is likely to increase the variation of the data access latency which also means the performance variation. Exceptions exist due their tiny cache footprints which fit in the L2 cache. However, such applications are rarely seen in popular web applications and thus we would not take them into consider in this paper. Although precise prediction/evaluation of cache allocation is highly needed for its great help to the performance analysis, we found few attentions were paid to the estimation/prediction of the cache allocation. Literatures [4–6] focused on the estimation of the total extra cache misses as a summative analysis of the performance. Some researchers [7] proposed phase-detecting-based model for the performance variation analysis. They employed a reuse distance-based cache interference estimation method which is in great complexity to profile and compute [8]. Other works [9–11] investigated cache partitioning from the aspects of both hardware and software. They aim to guarantee the worst performance by adopting fair cache sharing, where cache is inefficiently utilized and modification of hardware is required. From our point of view, the lack of achievements on the estimation/prediction method for cache allocation and model of performance variation put us at an inferior position to mitigate/ease the performance variation. As we stated above, the co-runner scheduling algorithm plays an important role in lowering the performance variation. However, most researches of scheduling policies [12–17] focused on the optimization of the overall performance or better energy efficiency without taking the performance stability into consideration. Based on the close relationship between cache allocation and performance stability, we are motivated to design a cache allocation prediction method which is of great help in the modeling of performance variation and the design of a performance-stability-oriented co-runner scheduling algorithm. The proposed co-runner scheduling algorithm can effectively improve the performance stability of the co-runners. The main contributions of this work are listed as follows:

- We propose a novel cache allocation prediction model with high parallelism which is convenient to be accelerated by GPU to predict the cache allocation of each co-runner. With a proper sampling interval, our prediction algorithm incurs low overhead when applied in runtime and meanwhile maintains sufficient details of the cache allocation variation which is valuable in the analysis of performance variation and the design of the co-runner scheduling algorithm.
- We deduce a performance-stability-oriented co-runner scheduling algorithm by leveraging the cache allocation information obtained under our cache prediction algorithm and mapping the co-runner scheduling target to a matching problem. The algorithm can solve the problem with a time complexity of $O(n)$.

The rest of this paper is organized as follows: in Sect. 2, we provide literature review as well as the motivation of our work. Section 3 is contributed to the description of our

eviction probability-based cache allocation prediction model and its parallelization. Section 4 proposes an performance-stability-oriented co-runner scheduling algorithm by utilizing the cache allocation information and several other important observations. We present evaluation results and explain the relationship between stability and performance based on our algorithms and observations in Sect. 5. Finally we give concluding remarks in Sect. 6.

We present evaluation results in and explain the relationship between stability and performance based on our algorithms and observations in . Finally we give concluding remarks in Sect. 7.

2 Related work and motivation

This paper investigates the problem of cache allocation prediction and the design of a performance-stability-oriented co-runner scheduling algorithm. Both of the issues have received relative few studies, for the former one, most of the previous works focused on the minimization of the overall performance degradation; and the researches of the co-runner scheduling policy were mostly aiming at the optimization of the overall performance/energy. We give a brief introduction of previous works as follows:

2.1 LLC allocation prediction

Previous researches on LLC allocation prediction are rarely seen. Most related works focused on the evaluation of the locality and use the locality metrics to estimate the cache inferences or total extra cache misses. Some works [5,6] established extra cache misses estimation models which take the summation of all possible combinations of memory access sequences. In [18], the authors simplified the evaluation of the locality by introducing a probability model and use this model to predict the cache inferences. The prediction of the cache allocation in runtime cannot benefit from such methods for they did not provide a clear algorithm to calculate the LLC allocation and the total extra cache misses is only calculated as a summative evaluation. Xu et al. [7] established a cache allocation estimation model by taking reuse distance as a probability factor. However, the method is very complicated in the profiling procedure and cannot be used as an online method to predict the cache allocation. Most importantly, an unrealistic assumption that they made for their methods was that the memory access are evenly performed on the cache sets while we found them highly imbalanced. As we stated above, the LLC allocation is the key to the analysis of performance variation. Due the lack of sound and practical methods for the prediction of LLC allocation, we are motivated to establish a practical yet accurate prediction method for the LLC allocation with a solid theoretical foundation. Since there are thousands of cache sets in an LLC and the memory access to them are not uniform, we design our cache allocation prediction method based on simple metrics which are easy to profile at runtime. Through this design, our algorithm can efficiently utilize the high parallelism of GPU. By specifying every cache set a prediction thread, we gather the cache allocations of each cache set and then further analyze the information.

2.2 Co-runner scheduling policy

Contemporary co-runner scheduling policy is load-balance oriented, which can possibly generate a co-runner mapping with poor performance. To address this problem, previous literatures have done a lot of work on the reduction of the overall performance degradation. Jiang et al. [13] mapped the co-runner selection target into a matching problem of an all-connected graph, and showed that the optimal solution is a schedule mapping with minimal summation of edge weights, which is an NP-hard problem. Based on the reuse distance, Xiang et al [19] use cache footprint and lifetime to measure the cache locality and use these information to direct the scheduling in hoping to achieve high overall performance. Cache interference reduction was also studied in [20] by pairing light and heavy tasks together. However, a theoretical analysis was absent and it is insufficient to deduce an accurate model by simply using cache miss rate. Among all these works, CRUISE[17] is a representative overall-performance-oriented algorithm that employs different policy to maximize the overall performance depending on the LLC replacement policy. They schedule the LLC thrashing programs together (for LRU) which could improve the overall performance. However, such policy will further exacerbating the LLC thrashing and the contention of memory bandwidth for those LLC thrashing programs. For example, scheduling the streaming applications together is very likely to incur increasing frame data delay with unpredictable high variation due to the bandwidth contention and the sharp change of LLC allocation while the other co-scheduled applications may run smoothly and the overall performance of the system looks good. Some other works are mainly fairness and worst case performance optimization focused. Among this type of scheduling policy, some researchers [9–11] present QoS analysis from the aspect of cache partitioning which aims to guarantee the performance of worst case at the cost of cache wasting. The main idea of fairness-oriented policy is *compensation*, i.e., allocate more cache space or prolong the CPU slice to the applications which shows poor performance [21,22]. Although this kind of scheduling policy may decrease the performance variation to some extent compared to the overall-performance-oriented policy, it is likely to incur resource waste and large overall performance degradation. As can be seen, previous works seldom pay attention to the optimization of the performance stability which, from our point of view, should be considered with more importance especially for the CMP severs running latency variation sensitive applications. In concern of the insufficient study that performance stability has received, we are thus interested in implementing a co-runner scheduling algorithm to guarantee a good performance stability (i.e., low variation). One may think that a performance-stability-oriented co-runner scheduling algorithm seems to tradeoff between performance and stability and thus its overall performance is poor. However, our evaluation shows that the average overall performance drop of our performance-stability-first mapping from the best overall performance mapping is within 7.8 % and with an average 17.5 % performance variation decrease. Additionally, a section is contributed to the discussion of the relationship between total performance and stability. With all the considerations above, our algorithm is designed to decrease the performance variation of the co-runners and obtain the optimal solution, i.e., a co-runner mapping, with a time complexity of $O(n)$.

3 Prediction of the LLC allocation

The cache allocation of each co-runner is an important factor to the analysis of cache contention from which the performance variation can be further modeled. An accurate prediction of cache allocation is of great help in directing the designment of a co-runner scheduling algorithm no matter it is overall-performance or overall-stability oriented. One possible method for the prediction of the cache allocation is by applying an Allocation-MissRatio curve [23], which is the sample of the (*cache allocation, miss ratio*) pair. Once the curve is obtained, the cache allocation can be obtained by referring to the curve. However, the main difficulty in the adoption of this method is that a specified cache allocation has to be allocated for a program which is very difficult to implement on a real machine. Thus the miss rate-cache allocation curve is only a rough reflection of the relationship between miss rate and cache allocation which is not accurate enough to be referred to. We try to solve the problem from a different perspective: the *eviction probability* (abbreviated as EP hereinafter), i.e., a co-runner's probability of being chosen as the victim of next eviction. If we can construct a metric which is positively correlated with EP with high accuracy, we can use it to approximate the EP and formulate a Markov Chain model for cache allocation prediction. The model can be clearly established by leveraging this metric (we call it EP although it can never be the exact eviction probability) and miss rate information obtained from Performance Monitor Counter (PMC). In the following, we first introduce an important metric, called allocation mean, with which EP can be derived.

3.1 Allocation mean

Assume there are two threads, denoted as i and j, contending for cache allocation. Without loss of generality, we assume that the hit rate of thread i is higher than that of thread j. Consequently, the data of thread i are very likely to be reaccessed before eviction and then relocated to the head of the LRU queue. However, for thread j, due to the low hit rate, few cache data will be reaccessed before eviction and most of its cache data will shift toward the tail of the LRU queue. In this scenario, we can deduce that in an LRU queue, once the contending threads are fully warmed up, the thread with a relatively high hit rate (thread i) will occupy the positions which are close to the head while the data of the thread with a relatively low hit rate (thread j) are squeezed toward the tail. As the gap between the hit rate of the two threads becomes larger, the separation of the data of these two threads in the LRU queue becomes more distinct.

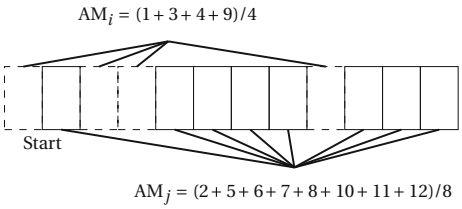
If any consecutive cache eviction occurs, most of the data to be evicted belong to thread j. However, as the allocation of thread j decreases, the tail region of the LRU queue is partially filled up with the data of thread i and EP_i increases. A stable state of the cache allocation can be eventually reached if the miss rate does not have a large variation. Intuitively, we want to find a metric to measure the extent of such separation which is an indication of the EP for the competitors. Despite of reflecting the relative position of each thread's data in the LRU queue with high accuracy, this metric should be easy to calculate since we will run it on GPU with high parallelism, i.e., unconditional add calculations are preferred. Finding such a metric will be our

first step to accurately predict the cache allocation. We use *Allocation Mean* (AM) as the metric to measure the extent of data separation. Figure 2 is an illustration of AM, defined as

$$AM_i = \frac{1}{n}\sum_{k=1}^{n_i} \text{pos}(d_k), \tag{1}$$

where AM_i is the allocation mean of thread i, n_i is the number of cache lines occupied by thread i, d_k is the kth cache line of thread i and pos is a function that returns the position of cache line i. As can be seen from Eq. (1), AM is the mean of the position indices of cache lines which belong to the same thread in the LRU queue. We illustrate the relationship between cache allocation and the allocation mean in Fig. 3 which we export from our LLC simulator(a memcached server co-running with an httpd server). It shows the cache allocation and the corresponding allocation mean of two threads sharing the LLC. Apparently there exist two clusters, one has a relative small cache allocation and a larger allocation mean while the other on the contrary. The reason for this clustering phenomena is just as we described above, i.e., a small value of AM indicates that the data are likely to be located close to the head of the queue while a large value of AM means that most of the data is located near the tail.

Fig. 2 The allocation mean



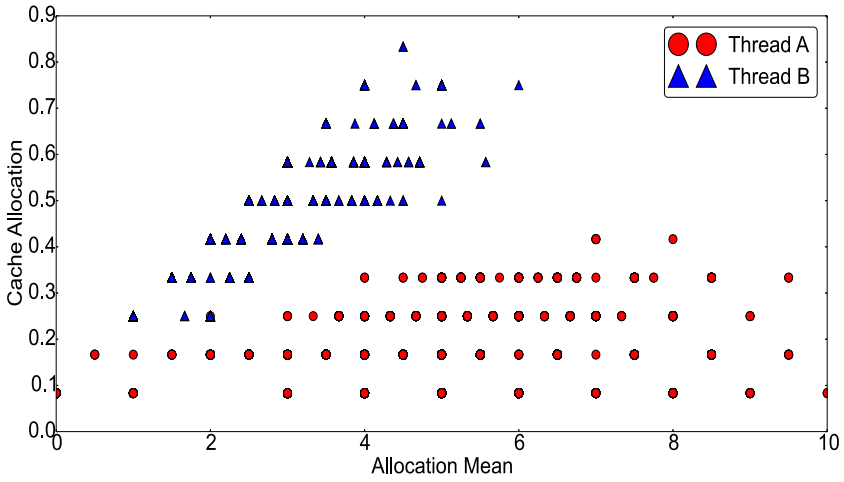


Fig. 3 The scatter plot of cache allocation–allocation mean point

3.2 Profiling allocation mean online

To acquire AM of each thread, we implement an instruction sampler based on PIN tools which can capture the memory access address. The sampling procedure is activated when sharp miss rate changes are monitored. The reason for not choosing phase change [24] as the trigger is that AM is only affected by the miss rate but in many cases miss rate changes while the phase not, especially in LLC-sharing context due to the effect of co-runners. The sampling procedure is suspended once 10,000 consecutive memory instructions are sampled. According to our experiment, the average IPC decrease caused by the sampling is 18.2 % and typically exits within 1 s. We estimate this sampling procedure fast and easy to proceed and is acceptable when applied online. Furthermore, an AM quick reference algorithm can be used to record and restore the AM when the periodicity of the miss rate is identified which can further decrease the sampling requirement. Such case is not rare when the co-runners have stable incoming workload. The memory access sequence is kept in memory read-only and then set as the input of a GPU-based LLC simulator written with CUDA tool kit. In our case, there are totally 8192 instances of device functions running in parallel with no need for synchronization and each is responsible for the memory access that fall into its corresponding cache set. Figure 4 is a normalized histogram of AM of an Httpd process sharing LLC with a Proftpd process. By performing a K–S test, we cannot reject the hypothesis that the AM obeys a normal distribution. Since there are thousands of cache sets, a statistical metric is the best way to depict the behavior of the cache contention. Therefore, we use the expectation of the distribution as the indicator of the Allocation Mean.

3.3 Estimation of the eviction probability

Although AM can roughly indicate the position of thread's data in LRU queue, EP cannot be fully reflected by simply using AM. See Fig. 5 for illustration. In the figure there are two threads sharing the cache set. Thread i has a larger cache allocation and

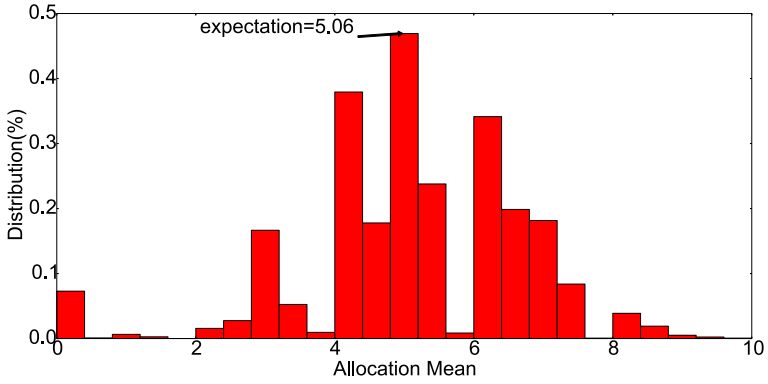


Fig. 4 AM histogram of an apache server

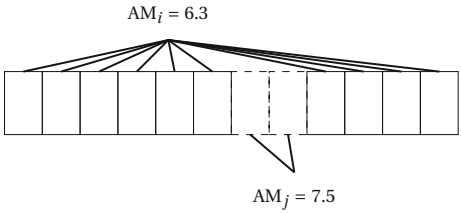


Fig. 5 AM and allocation size

has an AM value of 6.3 while thread j only holds 2 cache lines and has an AM value of 7.5. We may draw the conclusion that thread j is more likely to lost allocation if considering AM only. However, the conclusion does not hold due to the fact that, although thread i has a smaller value of AM, it has a much larger allocation which increases the EP (the larger the portion a thread occupies, the larger the probability that the data appear close to the tail of the LRU queue). For this concern, we take both AM and current data allocation into consideration when estimating the EP. Denoting C_i as the cache line count allocated for thread i and CACHEWAYS as the cache ways of the LLC, we have

$$AL_i = \frac{C_i}{\text{CACHEWAYS}} \tag{2}$$

Next, we denote by AM_i the expectation of thread i's Allocation Mean and EP_i as the *relative* eviction probability of thread i. We have the following equation

$$\text{EP}_i = \frac{\text{AM}_i}{\max(\text{AM})}\text{AL}_i \Big/ \sum_{k=1}^{n} \frac{\text{AM}_k}{\max(\text{AM})}\text{AL}_k \tag{3}$$

Since this definition take both the position and allocation information into account, it can reflect the relative probability of being chosen as the eviction victim.

3.4 Predicting the cache allocation

3.4.1 GPU-accelerated LLC simulator

For the purpose of online application, EP should be profiled quickly which means taking the advantage of the high parallelism of GPU is necessary. Using GPU to accelerate the LLC simulation has been seen in [25], the simulator is parallelized in cache set level and the search is also parallelized. However, in our implementation we do not need the bucket sort procedure (which incurs 36 % overhead). We do not parallelize the LRU search process neither for we found no prominent performance improvement by adopting the parallel search (due to the large synchronization overhead their algorithm needed).

3.4.2 Predicting the cache allocation

Once we obtain the EP of each thread, the contention of the cache allocation can be modeled. Assume threads i and j sharing the LLC. Denote $\text{MissRate}_i(t)$ as the LLC miss rate of thread i and Δ_i as the cache allocation variation of thread i. We have

$$\Delta_i(t) = \text{MissRate}_i(t-1) \times \text{EP}_j(t-1) - \text{MissRate}_j(t-1) \times \text{EP}_i(t-1) \quad (4)$$

Equation (4) implies that the cache allocation of thread i will change under two conditions: (a) Thread i encounters a cache miss and the victim cache line belongs to thread j. In this case the cache allocation of thread i will increase. (b) Thread j encounters a cache miss and the cache line belongs to thread i is chosen as the eviction victim. In this case, the cache allocation of thread i will decrease. We use libpfm to calculate $\text{MissRate}_i(t)$ by reading out the cache miss related event of thread i from PMC.

To comply with the LRU algorithm, $\Delta_i(t)$ needs to be rounded up into either 1 or -1 in each step of prediction. The round up function can be simply defined as

$$D(\Delta_i(t)) = \begin{cases} 0 & \\ -1 & \Delta_i(t) < 0 \\ 1 & \Delta_i(t) > 0 \end{cases} \quad (5)$$

Thus, the cache line count for next period $C_i(t)$ can be expressed by

$$C_i(t+1) = C_i(t) + D(\Delta_i(t)) \quad (6)$$

Assume that the two co-runners are fully warmed up and AMs are profiled, we present the pseudo-code of this algorithm in Algorithm 1. Be noted that this algorithm is responsible for single cache set; therefore, there are 8192 instances of this algorithm running in parallel on GPU in our case.

ALGORITHM 1: Cache Allocation Prediction

Input: MemroyTrace
Output: Cache Allocation
$t = 0$;
Set MissRate of each thread according to PMC **for** *Each MemoryTrace* **do**
  $\text{EP}_i(t+1) = \frac{\text{AM}_i(t)}{\max(\text{AM}(t))}\text{AL}_i(t) / \sum_{k=1}^{n} \frac{\text{AM}_k(t)}{\max(\text{AM}(t))}\text{AL}_k(t)$;
  $\text{EP}_j(t+1) = 1 - \text{EP}_i(t+1)$;
  if *cache not filled* **then**
    Continue ▷ Skip the prediction;
  end
  $\Delta_i(t) = MissRate_i \times \sum_{k=1}^{n} \text{EP}_k(t) - \sum_{k=1}^{n} MissRate_k \times \text{EP}_i(t)$;
  $\text{C}_i(t+1) = \text{C}_i(t) + \text{D}(\Delta_i(t))$ ▷ Update the Cache Allocation;
  $t = t + 1$;
end
Save Current EP;

The prediction procedure can be depicted as follows. First, we obtain the initial value of AM, AL and calculate EP for each thread. Then in each time period, for each thread, we calculate the cache allocation change according to Eq. (4). Once the changes are calculated, the cache allocation of each thread in the next time period can be predicted. Next, EP will be updated due to the change of cache allocation. To further reduce the overhead, the EPs are saved for the fast approximation in the next run. Updating miss rate every loop only introduces periodic randomness if the miss rate variation is white noise like and thus it is set as constant. This prediction procedure is activated when the AM is re-profiled as discussed above (i.e., a considerable miss rate change happens). During the interval, the results will be deemed as the cache allocation for this time period. According to our test, 200,000 memory instructions are sufficient to warm up the total 8192 cache sets of an 8 MB LLC. This amount of data can be processed within 0.7 seconds by our GPU-accelerated prediction algorithms, which we consider is acceptable during practical usage. Denote by t_o as the total overhead of our algorithm, t_{tr} as the overhead of trace collecting, t_{ker} as the overhead of the kernel function running on GPU. For memory trace within billions of records, t_{ker} will not exceed 6 seconds. However, t_{tr} may vary depending on the memory allocation and the OS scheduling which has a direct influence on the memory access rate and the contention of memory bandwidth.

We now demonstrate the accuracy of our algorithm from both single cache level set and the whole LLC level by showing the prediction curve.

Figure 6a presents the prediction curve of cache set 0 (for proftpd sharing LLC with httpd) generated from Algorithm 1 when the co-runners are fully warmed up. It can be seen from the figure that the prediction curve can well approximate the simulated value. Note that the minimal change for a 12-way cache sets is 8.3 % (one cache line), thus we say that a steady state is reached as long as a periodic oscillation is within 8.3 %. We can see that when encountering sharp miss rate variation, there exist several cycles delay for the predicted value. The delay is relative small, after

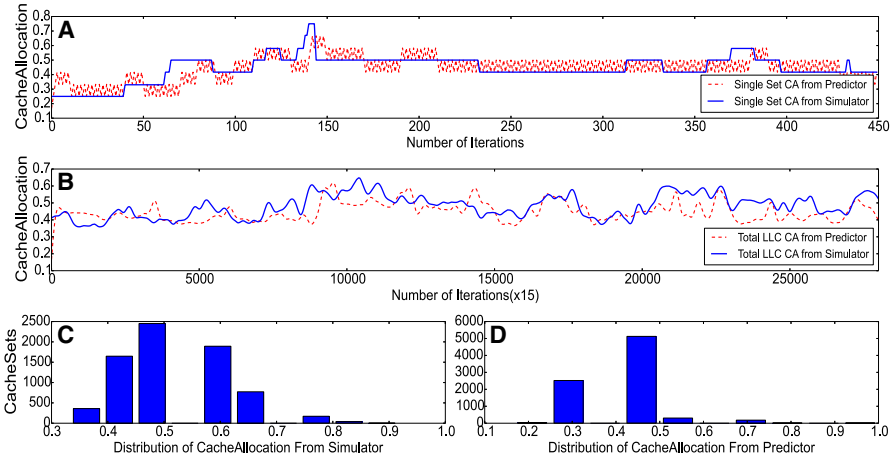


Fig. 6 Allocation prediction curve for a single cache set

all, if EP can truthfully reflect the relative eviction probability, the convergence will not take long time because the cache line count (typically 12) is small. In another aspect, we found that the predicted value is biased from the simulated one even in stable state,(i.e., the simulated value is not the mean of the oscillation in the flat part). Figure 6b shows the prediction curve of the entire LLC, i.e., totally 8192 cache sets, the two curves are the averaged values of the counterpart in (a). It takes more iterations to converge than the single cache set case for the steady state can only be reached when a large portion of the cache sets are fully visited. In some part, the bias exists (from $\times$ ticks 2000 after), the reason could be the non-normal distributed bias in the single cache prediction. The source of such bias is the inaccuracy of the miss rate. For in our algorithm, the miss rate is used as the same through all cache sets which in fact brings in the error. If the cache miss happens extreme unevenly (e.g., most cache miss happened on few cache sets), then in this time period, the predicted cache allocation is biased and this motivated us to represent the prediction result in the form of distribution instead of the average of summation, which can demonstrate the evaluation more clearly. Figure 6c, d shows the cache allocation histogram of totally 8 K cache sets at time t from the simulator and the predictor, respectively. The expectation curve is presented in the evaluation section. We verify that the two cache allocation distributions obey a positively skewed normal distribution and their expectations are very close. In Fig. 6d, the distribution of the cache allocation exhibits a smaller variation, which is caused using the averaged MissRate for every prediction instance (and thus the summation is not the same which result in the bias). From another point of view, this weakens the distinction between cache sets and thus make the value more concentrated to the expectation. In other words, our prediction algorithm is apt to predict the cache allocation for *most* cache sets instead of the total averaged. Now with an accurate prediction of the cache allocation for the majority of cache sets, we can foresee and estimate the performance variation. To present our prediction method more clearly, Fig. 7 shows the overview of our prediction method with every critical data path and function unit. Next we introduce a co-runner scheduling algorithm by applying our new metric, i.e., EP. With its performance-stability-oriented scheduling policy, the performance variation of the system will decrease when facing the changing environment, e.g., sharp miss rate variation caused by increasing workload, phase change, etc. We limit the number of threads that sharing the LLC at two which is sufficient to clarify our principle, the algorithm can be greatly complicated when the sharing number is over two, which we would like to leave it as our future work.

4 A performance-stability-oriented co-runner scheduling algorithm

As aforementioned, the performance is positively correlated with cache allocation in most cases. Hence, to design a performance-stability-oriented co-runner scheduling policy is equivalent to implement an algorithm under which the cache allocations of the co-runners suffer small variations. After careful consideration, our algorithm is designed to find an optimal mapping which can guarantee a maximum number of co-runners with good performance stability and a minimum total performance variation for those which are not. By formulating a proper target function and a simple yet solid

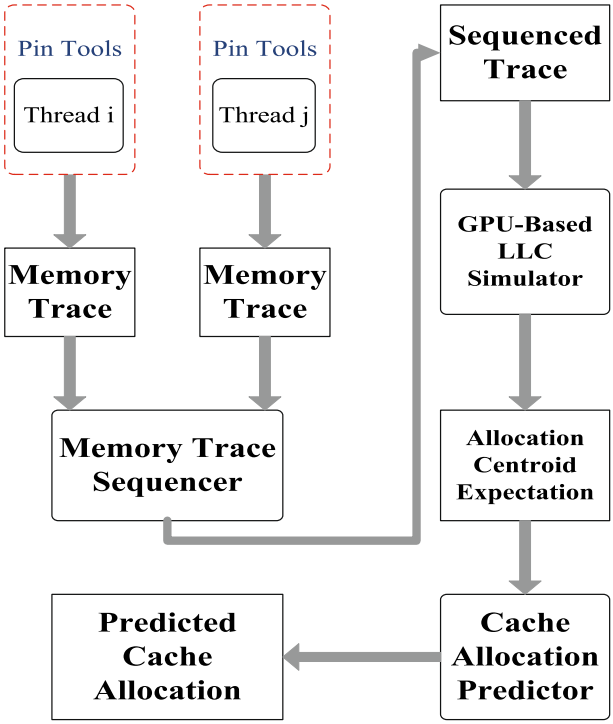


Fig. 7 Overview of the cache allocation prediction method

matching logic, our algorithm can find the optimal solution with a time complexity of $O(n)$.

4.1 The co-runner scheduling policy

Intuitively, one may speculate it reasonable to schedule programs with relative close EPs to sharing the LLC. We prove the rationality of this speculation through the following formulas. Denote $\mathrm{AL}_i^x(t)$ as the cache allocation of thread i under co-runner mapping x at time t and σ_i^x as the standard deviation of $\mathrm{AL}_i^x(t)$. If $\sigma_i^x < \sigma_i^y$, we say that AL_i is more stable under mapping x than y, i.e., we prefer mapping x to y. Assume in time t, the changes of miss rate are $\Delta\mathrm{MissRate}_i$ and $\Delta\mathrm{MissRate}_j$. According to Eq. (4), the difference of the cache allocation is:

$$\Delta_i(t+1) = \Delta\mathrm{MissRate}_i \times \mathrm{EP}_j(t) - \Delta\mathrm{MissRate}_j \times \mathrm{EP}_i(t) \tag{7}$$

Note that from the definition of EP, we know that $\mathrm{EP}_j(t) + \mathrm{EP}_i(t) = 1$. Assume $\Delta_i(t+1) > 0$. When $\mathrm{EP}_j(t)$ increases with a, we have the following inference

$$\begin{aligned} &\Delta\mathrm{MissRate}_i \times (\mathrm{EP}_j(t) + a) - \Delta\mathrm{MissRate}_j \times (\mathrm{EP}_i(t) - a) \\ &\quad > \Delta_i(t+1) \end{aligned} \tag{8}$$

Similar inference can be drawn that when $\Delta_i(t+1) \leq 0, \mathrm{EP}_j(t)$ decreases. On the contrary, when $\Delta_i(t+1) < 0$ and $\mathrm{EP}_j(t)$ increases, $|\Delta_i(t+1)|$ will be decreased. From our observation, generally the curves of $\Delta\mathrm{MissRate}_i(t)$ and $\Delta\mathrm{MissRate}_j(t)$ are twisted and varying which means all the two situations, i.e., $\Delta_i(t+1) > 0$ and $\Delta_i(t+1) \leq 0$ happens alternately and thus the variation of $\Delta_i(t)$ will be enlarged. With the following definition

$$\delta_{\mathrm{EP}}(j,i) = \left|\mathrm{EP}_j(t) - \mathrm{EP}_i(t)\right| \tag{9}$$

we can draw the conclusion that under the same miss rate change, a larger $\delta_{\mathrm{EP}}(j,i)$ is more likely to incur a larger cache allocation variation and the variation itself will also yield a large variance, that is, Δ_i and $\delta_{\mathrm{EP}}(j,i)$ are positively correlated. Therefore, to achieve performance stability, it is wise to schedule the threads with small $\delta_{\mathrm{EP}}(j,i)$, i.e., close EPs together.

4.2 The EP queue

To compare the EP of arbitrary two co-runners easily, we pick up a pivotal program and then all other programs are co-scheduled with this pivotal program to profile the EP. Given co-runner set S_c, we use the following steps as the first part of our scheduling algorithm, as shown in Algorithm 2. The results are sorted in ascending order as Q_{EP}. A heuristic obtained from the previous sections is that we need to pick out the co-running pairs by selecting the adjacent co-runners in Q_{EP} with the smallest $\delta_{\mathrm{EP}}(j,i)$. We now construct a weighted complete graph by taking advantage of Q_{EP}, which is of great help in solving the problem.

ALGORITHM 2: Performance Stability Oriented Co-Runner Scheduling Algorithm **Profiling EP**

Input: co-runner set S_c
Output: EP Queue Q_{EP}
Randomly pick one co-runner from S_c, mark as t_s;
for *each co-runner* t_i *in* S_c **do**
    Schedule t_i to co-run with t_s;
    Profile EP_i;
    Push EP_i into Q_{EP};
end
Push $\mathrm{EP}_s = 0.5$ into Q_{EP};
Sort Q_{EP} in ascending order;

4.3 The EP graph

We map the elements in Q_{EP} into a weighted all-connected graph denoted as G_{EP} by labeling the vertices (i.e., the co-runners) with their position in Q_{EP}. For example, if co-runner a is the 4th element in Q_{EP}, then in G_{EP}, its corresponding vertex is labeled 4. The edge weight of G_{EP} is defined as

$$\omega_{i,j} = \begin{cases} \omega_{j,i} & \\ \delta_{\mathrm{EP}}(i,j) & i,j \neq n \\ 0 & j = n \end{cases} \tag{10}$$

We set an empirical threshold T_{ps} below which the two co-runners are said to be sufficiently close to each other and thus yield a small performance variation. In contrast, the co-running pairs whose $\delta_{\mathrm{EP}}(j,i)$ exceeds the threshold are considered to suffer high performance variation. For this reason, we call a co-runner pair whose $\delta_{\mathrm{EP}}(j,i) \leq T_{ps}$ is a *stable* pair, otherwise it is an *unstable* pair. Next we classify the edges into two categories, i.e., the *Red Edge* and the *Black Edge*. The edge is colored red only if its weight is no larger than T_{ps}. Otherwise it is colored black. The subgraph composed of red edges is called a Red Graph denoted as G_R. Similarly we denote by G_B the subgraph called Black Graph, of which the edges are all black.

Instead of targeting for the minimization of the total performance variation. The algorithm should try to find a co-runner mapping which can maximize the number of co-runner pairs with small performance variation and minimize the total performance variation of the unstable pairs. This target holds more practical meaning than finding a total minimum performance variation for the reason that there may exist a co-runner mapping with a minimal $\sum_{i=1}^{n} \Delta_i$, yet most of the performance variation decrease is caused by a single program while other programs may exhibit little performance variation decrease. Furthermore, it will be more appealing to the operation of network program server for that a maximum number of performance-stable co-runner pairs can be beneficial to more services.

Now our target is transformed to a matching problem, i.e., select $n/2$ edges with no any two edges sharing the same vertex, the optimal solution is the one that contains most red edges and meanwhile yields the minimum summation of weights of black edges. We introduce our algorithm by taking a 10-vertex case, i.e., schedule 10 programs for co-running, as an example. Figure 8 represents the EP distance matrix (i.e., the weight matrix of G_{EP}) deduced from Q_{EP} and set T_{ps} as 0.1. G_{EP} is shown in Fig. 9, where the black edges are not drawn for the clarity of vision.

4.4 Pairing the vertexes

First we introduce Lemma 1 and Lemma 2 provides two important attributes to the EP Graph, which are the keys of our algorithm.

Lemma 1 *In EP Graph G_{EP}, for three arbitrary vertices i, j and k that satisfy $k-i > j-i$, we have $\omega_{i,k} > \omega_{i,j}$*

Proof Since Q_{EP} is an ordered queue, according to Eq. (10), we can deduce that $\omega_{i,k} > \omega_{i,j}$ if k is a larger index than j in Q_{EP}. □

Lemma 2 *Let H be the ordered set of the vertexes in a Red Graph. Then H is a set of consecutive integers.*

	1	2	3	4	5	6	7	8	9	10
1	**0**	0.2	0.23	0.3	0.35	0.46	0.57	0.6	0.67	0.84
2	0.2	**0**	**0.03**	**0.1**	0.15	0.26	0.37	0.4	0.47	0.64
3	0.23	0.03	**0**	**0.07**	0.12	0.23	0.34	0.37	0.44	0.61
4	0.3	0.1	**0.07**	**0**	**0.05**	0.16	0.27	0.3	0.37	0.54
5	0.35	0.15	0.12	**0.05**	**0**	0.11	0.22	0.25	0.32	0.49
6	0.46	0.26	0.23	0.16	0.11	**0**	0.11	0.14	0.21	0.38
7	0.57	0.37	0.34	0.27	0.22	0.11	**0**	**0.03**	**0.1**	0.27
8	0.6	0.4	0.37	0.3	0.25	0.14	**0.03**	**0**	0.07	0.24
9	0.67	0.47	0.44	0.37	0.32	0.21	**0.1**	**0.07**	**0**	0.17
10	0.84	0.64	0.61	0.54	0.49	0.38	0.27	0.24	0.17	**0**

Fig. 8 EP distance matrix

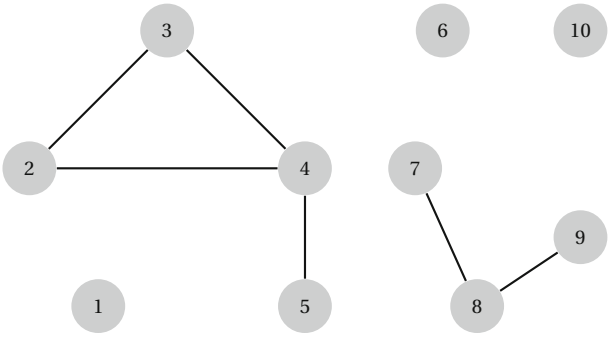


Fig. 9 The EP graph

Proof We prove by contradiction. Assume there exist h_i and h_{i+1} in set H, and they satisfy $h_i - h_{i+1} > 1$. Let $h_i - h_j = 1$. Then $h_j \notin H$. According to the definition of G_R and G_B, we have $\delta_{\mathrm{EP}}(h_j, h_i) > T_{ps}$ and $\delta_{\mathrm{EP}}(h_i, h_{i+1}) < T_{ps}$. This contradicts with Lemma 1. Therefore, H is a set of consecutive integers. □

In G_R, there could exist several connected subgraphs. According to the definition, the vertices in the subgraphs can be paired sequentially without generating an unstable one. For G_B, we implement a linear scanning method to guarantee that the summation of the edge weight is the minimal by leveraging Lemma 1. Algorithm 3 presents the corresponding pseudo-code.

ALGORITHM 3: Performance-Stability-Oriented Pairing Algorithm **Matching**

Input: EP Graph G_{EP}
Output: Co-runner mapping
$V_{idx} = 1$;
G_{CG}= Connected SubGraph contains $V_{idx} \in G_{EP}$;
$V_{unmatched} = null$;
while $G_{EP} \neq$ NULL **do**
  if NUM(G_{CG})%2 **then**
    if (NUM(G_{CG}) == 1) **then**
      Push($V_{unmatched}$, G_{CG});
    else
      if $V_{unmatched}$%2 **then**
        $G_{CG} = G_{CG} -$ MIN_INDEX(V_{idx});
        Push($V_{unmatched}$, MIN_INDEX(V_{idx}));
      else
        $G_{CG} = G_{CG} -$ MAX_INDEX(V_{idx});
        Push($V_{unmatched}$, MAX_INDEX(V_{idx}));
      end
      Pair $< t_i, t_{i+1} > \in G_{CG}$ sequentially;
    end
  else
    Pair $< t_i, t_{i+1} > \in G_{CG}$ sequentially;
  end
  $G_{EP} = G_{EP} - G_{CG}$;
  V_{idx} = MIN_INDEX(EP);
  G_{CG}= Connected SubGraph contains $V_{idx} \in G_{EP}$;
end
Sequentially pair vertices in $V_{unmatched}$;

As can be seen in Algorithm 3, in the first loop, G_{CG} containing vertex 1 is identified. Denote it as G_1, since G_1 has only one vertex 1, it will be pushed into the unmatched vertex set $V_{\text{unmatched}}$. Then in the second loop, G_{CG} that contains vertices 2, 3, 4, 5 is identified. Since the number of vertices is even, according to the algorithm, the vertices will be sequentially paired. Then 6 is identified as a G_{CG} in loop 3, and pushed into $V_{\text{unmatched}}$. In the 4th loop, 7, 8, 9 are the vertices contained in G_{CG}, where vertex 9 has the largest index according to the Q_{EP} order and is pushed into $V_{\text{unmatched}}$ while 7, 8 is scheduled to run together. In the last loop, vertex 10 is the only element in G_{CG} and pushed into $V_{\text{unmatched}}$. Note that G_{CG} is removed from G_{EP} at the end of each loop. Now $V_{\text{unmatched}}$ contains 1, 6, 9, 10, and will be sequentially paired. The final schedule mapping of this case is $\{\langle 1, 6\rangle, \langle 2, 3\rangle_{ps}, \langle 4, 5\rangle_{ps}, \langle 7, 8\rangle_{ps}, \langle 9, 10\rangle\}$, which contains three performance stable pairs (subscripted with ps) and two performance unstable pairs. We call this algorithm *Sequential Connected Sub-Graph Identifying and Pairing*. Using Lemmas 1 and 2, we can prove that our algorithm can achieve a total minimal summation of black edges as well as the maximum number of stable pairs. Another advantage of this algorithm is that it has a time complexity of $O(n)$. As we can see from the pseudo code, there is only one loop which iterates all vertices, meaning that the algorithm can accomplish within exactly one scan of the vertices.

4.5 Scale-up of the algorithm

With the advent of quad-core and octa-core CMPs in recent years, a co-runner scheduling algorithm should be capable of scheduling four or eight programs to share the LLC and yield small performance variation. However, the problem cannot be described properly under the 2-D EP graph and is proved to be an NP-hard problem. The cost for achieving an optimal solution is expensive and thus we can use a recursive way to solve this problem, that is, using EP Graph-based algorithm to do the first round mapping. Then using the weighted average EP to represent the co-runner pairs, i.e., the two co-runners are merged as one. Then the EP Graph is regenerated and our algorithm applied again. For quad-core cases it will take two rounds and three for octa-core. This is obviously a sub-optimal solution yet still the easiest and most practical one. We skip further details for the page limitation.

5 Experimental evaluation

In this section, we present the evaluation of the proposed LLC allocation prediction model and the performance-stability-oriented co-runner scheduling algorithm. Our testbed is a server with two Xeon X5770 Octa-Core CPU on board. Xeon X5770 has two LLC-sharing domains with each one composed of four dual-thread cores. There are eight cores sharing an 8 MB LLC in each domain. Each program is accommodated with four cores. See Fig. 10. Thus, each domain will have two programs sharing the LLC and the eight applications running on the server simultaneously. We choose eight applications along with their profiled EP as shown in Table 1. These applications are mainly typical IO-bounded web applications except several computation bounded ones. The reason for choosing both kinds of applications is that we want the EP gap to be enlarged which can demonstrate the effect of our algorithm better (as you can see the web applications do not show large EP gap essentially). Note that, Lucy is selected as the pivotal program in the profiling of the EP. In our experiment, we use standard test client if it is provided, otherwise the client is implemented by us. The

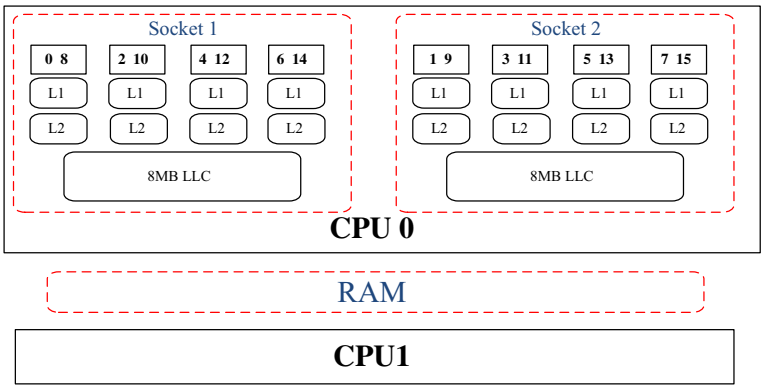


Fig. 10 Memory hierarchy of testbed

Table 1 EP distances

Application	Test client	Index	EP
Httpd server	http_load	1	0.77
Proftpd server	Proftpd client	2	0.70
Postfix	Author implemented	3	0.69
Subversion	Subversion client	4	0.65
Memcached	memslap	5	0.61
Lucy	Author implemented	6	0.5
MLPack	knn-search example	7	0.33
XgBoost	LR demo	8	0.29

service requests frequencies are in the form of periodic step functions with different period and are controlled by expect script.

5.1 Evaluating cache allocation prediction

We evaluate our algorithm by comparing the expectation of cache allocation from the predictor and the simulator. As is explained in Sect. 3, we gather all the 8192 predicted cache allocation values for the entire LLC and obtain the expectation by fitting them with normal distribution. The curve of the expectation is what we presented. We first skip a 200,000 warm-up memory instructions, and then dump over 120,000,000 memory instructions with their corresponding cache line number time stamp for every co-runner as the initial profiling.

We pair every two programs to share the LLC. Figure 11 shows the prediction and simulation curves of the cache allocation expectations. Four co-runner pairs are shown, i.e., ⟨MLPack, Memcached⟩, ⟨MLPack, Httpd⟩, ⟨Xgboost, MLPack⟩ and ⟨Memcached, Httpd⟩. We skip the presentation of other cases due to page limitations. It can be seen that there is a drastic variation of the amount of instructions that our algorithm needs to take to achieve a stable prediction value under different co-runner mappings. In addition, the shapes of the prediction curves are quite different under the four co-runner pairs. Specifically, the one obtained under ⟨MLPack, Memcached⟩ pair looks more like a step function while the one under ⟨MLPack, Httpd⟩ pair remains almost as a straight line. The differences above are highly correlated with the memory access pattern (e.g., the reuse distance, etc.) of the programs.

Figure 12 presents the average accuracy under both low (suffixed with L) and heavy workload (suffixed with H) for three co-runner pairs. We found that under heavy workload, our predictor has a performance of smaller average errors and variation comparing to the low workload case. A reasonable explanation is that a heavy workload may lead to a higher memory reference rate, which can make our memory reference trace more *tight* which can lead to a more accurate simulation of the LLC; on the contrary, an idled application is likely to increase the randomness of the simulation. It also implies that our algorithm works more favorably under a heavy workload condition.

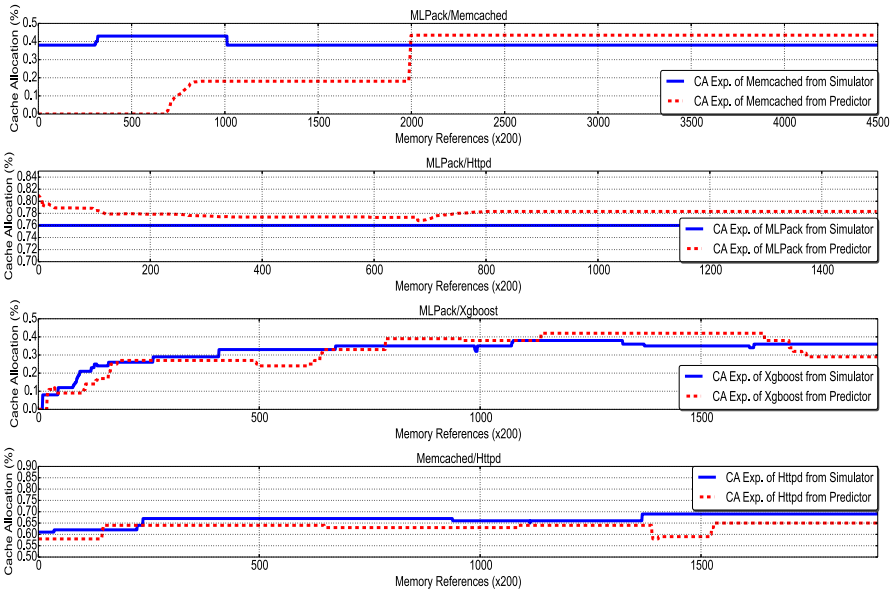


Fig. 11 Expectation curve of cache allocation

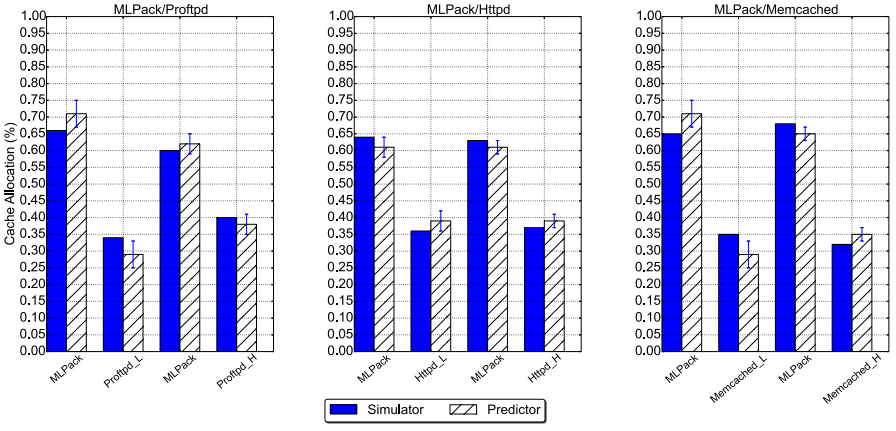


Fig. 12 The effect of workload variation

The average prediction error of all the 28 co-runner pairs (for each pair we present the prediction error for one of the two programs) are shown in Fig. 13. The average prediction error is around 5.2 % while the largest error is 11.3 %.

5.2 Evaluating performance-stability-oriented scheduler

To evaluate the effectiveness of our stability-oriented co-runner scheduling algorithm, we estimate the performance variation by the coefficient of variation which is widely

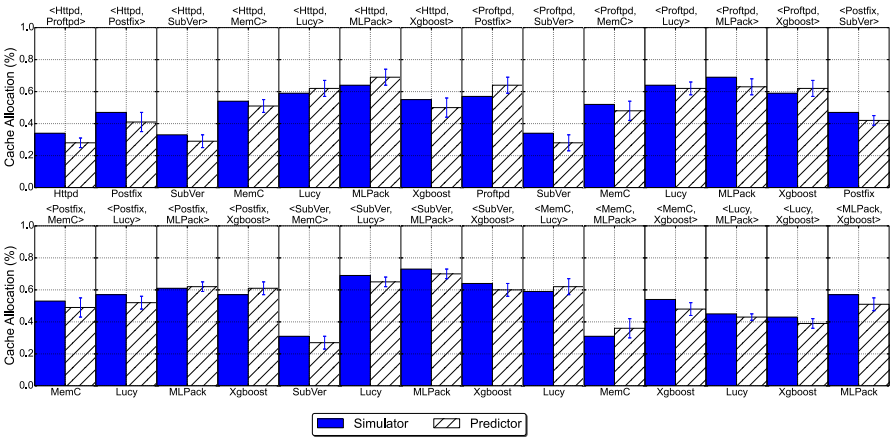


Fig. 13 The result of cache allocation prediction

used in time series analysis. Specifically, we denote by $CV_{j,i}$ the coefficient of variation of program j's IPC under scheduling mapping i. Then we have

$$CV_{j,i} = \frac{\sigma_{j,i}}{\mu_{j,i}} \tag{11}$$

where $\sigma_{j,i}$ is the standard deviation, $\mu_{j,i}$ represents the mean value. Let S_i be the stability score, defined in Eq. (12), as the variation metrics of scheduling mapping i.

$$S_i = \sum_{j=1}^{N} \alpha_j CV_{j,i} \tag{12}$$

α_j is the weight of each application and $\sum_{j=1}^{N} \alpha_j = 1$. Consequently, a better mapping is expected to yield a relative small S. We emphasize that the optimal solution in our algorithm does not necessarily have the smallest S , for there could be one pair with very low CV while others remain high. Recall the design of our algorithm is to maintain as many as possible co-runner pairs yielding a low CV (performance-stable state) when the miss rate changes. Figure 14 shows the CV under different workload for several co-runner mappings with four programs, which is sufficient to show the difference.

MLPack [26](kNN-Search) is a computation-bounded program and thus it is insensitive to the workload variation relative to other co-runners. We can see that under heavy workload, the CV of all co-runners increase in all mappings. The main reason that the co-runners have wider miss rate ranges under heavy workload is due to the severer bandwidth contention than that under low workload case, and thus the variation of miss rate increases, which, according to our theory, will increase the performance variation. From the slope of the lines, it can be seen that the CV increase rates of mapping 2 and 3 are larger than 1. Co-runner mapping 1 also has the smallest S increase rate. These two facts comply with the conclusion that we draw in our scheduling model, i.e., a

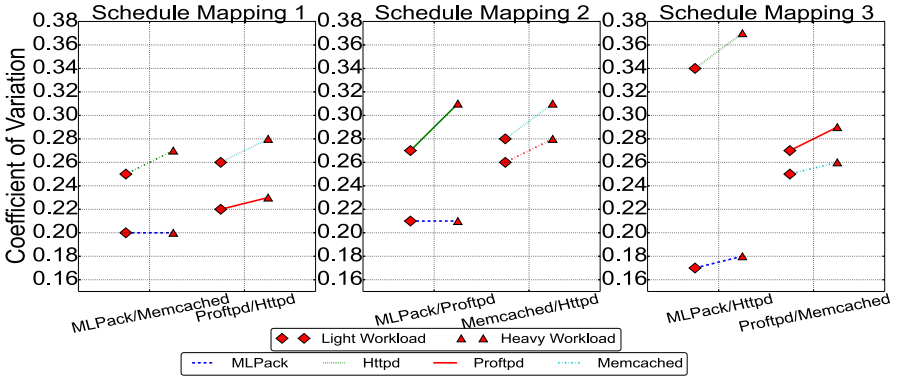


Fig. 14 Standard deviation of service latency

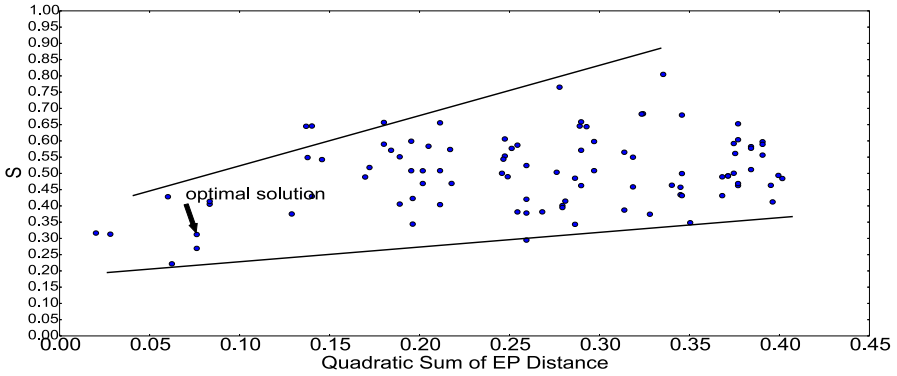


Fig. 15 Evaluation of EP distance and the corresponding S

smaller $\delta_{\mathrm{EP}}(k, i)$ of scheduling mapping may reduce the performance variation under varying environment.

There are totally 315 co-runner mappings for our case, see Table 1, due to the limitation of resources, we run 105 of them (including the optimal mapping generated by our algorithm) for 10 times. The scatter plot of S_i is also shown in Fig. 15. The X-axis represents the quadratic sum of the EP distance while the Y-axis is the S value. The reason for not choosing the sum of EP as X-axis is that lots of mappings have close sum of EP distance but relative large CV due to the existence of one co-runner pair with large EP distance. The quadratic sum of EP distances can reflect such difference and can scale up the X-axis for the purpose of a clear presentation.

We can see that the mappings with lower quadratic EP distance sum is located at the lower part of the figure and vice versa. The positive correlation between S and quadratic sum of EP distance can be easily identified from the figure(although the points are not so dense). With the growing of x axis, the max value of S increases significantly while the min value increases slowly. As is shown, the optimal solution does not yield the smallest S nor the smallest quadratic sum, however it has the most co-runner pairs running at a relative performance stable state and smallest EP distances

Table 2 Stability score

Mapping	Score	IPC	Mapping	Score	IPC
⟨2,3 | 4,5⟩ ⟨1,6 | 7,8⟩	0.312	0.392	⟨1,2 | 3,4⟩ ⟨5,6 | 7,8⟩	0.341	0.377
⟨1,5 | 2,6⟩ ⟨3,7 | 4,8⟩	0.781	0.421	⟨2,3 | 4,5⟩ ⟨6,7 | 1,8⟩	0.577	0.407
⟨1,6 | 4,5⟩ ⟨2,3 | 7,8⟩	0.291	0.349	⟨5,6 | 3,4⟩ ⟨1,2 | 7,8⟩	0.389	0.401
⟨3,7 | 2,6⟩ ⟨1,5 | 4,8⟩	0.797	0.442	⟨6,7 | 4,5⟩ ⟨2,3 | 1,8⟩	0.502	0.409
⟨1,8 | 2,7⟩ ⟨3,6 | 4,5⟩	0.591	0.401	⟨1,4 | 2,3⟩ ⟨5,8 | 6,7⟩	0.679	0.414
⟨3,6 | 2,7⟩ ⟨1,8 | 4,5⟩	0.603	0.399	⟨5,8 | 2,3⟩ ⟨1,4 | 6,7⟩	0.712	0.428

for those are not stable. With the modification of the EP threshold, the optimal solution can be different.

Especially, the S of some mappings which we are interested in are listed in Table 2. The last two rows are the same mappings with the first two rows, only their co-runner pairs running on different CPUs (every <> means a CPU). The programs are presented by their index in Table 1.

Our algorithm chooses ⟨Proftpd, Postfix⟩s, ⟨Subversion, Memcached⟩s, ⟨MLPack, Xgboost⟩s, ⟨Httpd, Lucy⟩ as the optimal solution when we set the EP threshold as 0.05. It does not have the smallest S but has the most co-runner pairs running at a relative stable state and the unstable pairs with the smallest CV comparing to other mappings. One notable observation is that the memory bus sharing does affect the CV, the first element in the third row shows slight improvement of stability than its counterpart at the first row. The following may explain this observation: the relative small CV of IPC could lead to a lower bus transfer variation, i.e., the need for bandwidth is relative stable during the same period. Assume there are four co-runner pairs, and two with low S, the other two with high S, our hypothesis suggests locating the co-runner pairs with low and high S on the same CPU (sharing the memory bus). For this can avoid the worst case of memory bus contention between the co-runner pairs with high S. Such relief of memory bandwidth contention feedback to the IPC variation decreases the CV. This improvement needs more theoretical deduction and equation to represent and we would like to leave it as our future work.

5.3 Evaluation of the overall performance

As we stated above, the trade-off between the overall performance and stability has to be acceptable. We compared the overall performance between stability-oriented algorithm, fairness-aware and overall-performance-oriented algorithms, respectively. We use average IPC, i.e., $T = \sum IPC_i/N$ as the metric of overall performance. We compare the performance between these three algorithms. According to the performance-oriented algorithms [14] described in Sect. 2, we profiled every combination of two programs and the optimal co-runner mapping generated by their algorithm is ⟨Proftpd, Postfix⟩, ⟨Subversion, Memcached⟩, ⟨MLPack, Xgboost⟩, ⟨Httpd, Lucy⟩. Then we applied fairness-aware methods, e.g., CPU slice prolonging on this mapping, the result shows that the S dropped with 6.7 % with T also dropped with 11.5 %. While

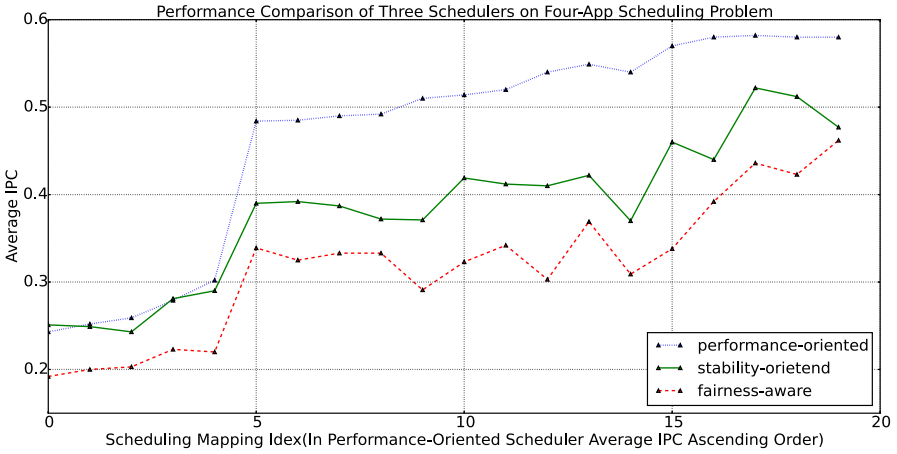


Fig. 16 Performance comparison between three schedulers

the schedule mapping generated by our algorithms has an S and T decreased 9.4 and 5.9 %, respectively. The result shows that the overall performance of our algorithm is 5.6 % higher while with nearly 3 % performance variation drop comparing to the fairness methods.

To be more statistically persuasive, we further divide the eight applications into four-application combinations to increase the evaluation samples. We randomly selected 18 application combinations (among total $C(8,4)/2$ ones). The results are shown in Fig. 16. The x-axis is the index of the combinations in ascending order of average IPC under performance-oriented scheduler and y-axis is the average IPC. We found that under low IPC (i.e., all four applications are computation bounded) situations, the average IPC of performance-oriented and stability-oriented mappings are close. With the average IPC getting higher, the performance-oriented scheduler always choose the mapping with highest IPC which enlarges the performance gap. It is rational to consider the stability orientation as an approximation of the performance-oriented scheduler when most of to-be-scheduled applications are computation bounded. The difference becoming more apparent when both none-computation-bounded and computation-bounded applications are to be scheduled. We can see that, comparing to the best performance, the stability-oriented mapping has an average 15.8 % drop of average IPC.

On the other hand, the fairness-aware algorithm needs to cooperate with an dynamic-strategy scheduler instead of employing an overall-performance-oriented scheduler (as previous works do) to maximize its benefit. The control of fairness must be dynamic with very quick response to avoid the resource waste, any mismatch (e.g., cache allocation, CPU slice) is likely to incur a rising performance variation and performance drop.

6 Conclusions

In this paper, we introduce a practical method for the prediction of cache allocation which takes the advantage of our novel metrics *Allocation Mean* and its derivation

Eviction Probability which reflect the relative probability of eviction. Due to the simplicity in the computation and high parallelism the prediction method is easy to apply. We further propose a performance-stability-oriented co-runner scheduling algorithm which targets to maximize the number of performance-stable co-runner pairs and minimize the total performance variations of the unstable ones. The algorithm uses EP as a key metric to map the selection of co-runners into a matching problem of a weighted all-connected graph. By leveraging the concept of EP distance and a proper optimization target, our algorithm can find the optimal solution with a time complexity of $O(n)$. Most importantly, the positive relation between EP distance and the CV of IPC can be identified from the evaluation which proved the effectiveness of our stability-oriented algorithm. The overall performance comparison between our algorithm and typical overall performance-oriented and fairness-oriented ones indicates that it is reasonable to say that our stability-oriented co-runner mapping has a overall performance between overall-performance-oriented and fairness-oriented schedulers meanwhile yielding the lowest IPC (i.e., throughput) variation.

Acknowledgments This work was supported by Huawei Innovation Research Program(HIPRO, Grant Number. YB2015080028).

References

1. Eyerman S, Eeckhout L (2008) System-level performance metrics for multiprogram workloads. IEEE Micro 28:42–53. doi:10.1109/MM.2008.44
2. Cazorla FJ, Knijnenburg Peter MW, Sakellariou R, Fernandez E, Ramirez A, Valero M (2004) Predictable performance in SMT processors. In: Proceedings of the 1st conference on computing frontiers. ACM, New York, pp 433–443. doi:10.1145/977091.977152
3. Jiang Y, Shen X (2008) Exploration of the influence of program inputs on cmp co-scheduling. Euro Conf Parall Comp. 263–273. doi:10.1007/978-3-540-85451-7_29
4. Sandberg A, Sembrant A, Hagersten E, Black-Schaffer D (2013) Modeling Performance Variation Due to Cache Sharing. In: Proceedings International Symposium High Performance Computer Architecture (HPCA), pp 155–166. doi:10.1109/HPCA.2013.6522315
5. Chen Xi E, Aamodt Tor M (2012) Modeling cache contention and throughput of multiprogrammed manycore processors. IEEE Trans Comp 61:913–927. doi:10.1109/TC.2011.141
6. Chandra D, Guo F, Kim S, Solihin Y (2005) Predicting inter-thread cache contention on a chip multiprocessor Architecture. In: Proceedings of International Symposium High-Performance Computer Architecture (HPCA), pp 76–86. doi:10.1109/HPCA.2005.27
7. Xu C, Chen X, Dick Rober P, Mao Zhuoqing M (2010) Cache contention and application performance prediction for multi-core systems. In: International Symposium Performance Analysis of Systems and Software (ISPASS). pp 76–86. doi:10.1109/ISPASS.2010.5452065
8. Xiang X, Ding C, Luo H, Bao B (2013) HOTL: a higher order theory of locality. In: Proceedings of the 18th Intl' Conf on Architectural support for programming languages and operating systems (ASPLOS '13). pp 343–356. doi:10.1145/2451116.2451153
9. Kim S, Chandra D, Solihin Y (2004) Fair cache sharing and partitioning on a chip multi-processor architecture. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques. pp 111–122. doi:10.1109/PACT.2004.15
10. Qureshi MK, Patt YN (2006) Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: Proceedings of Annual International Symposium on Microarchitecture (MICRO), pp 111–122. doi:10.1109/MICRO.2006.49
11. Suh GE, Devadas S, Rudolph L (2002) A new memory monitoring scheme for memory-aware scheduling and partitioning. In: Proceedings of International Symposium on High Performance Computer Architecture. doi:10.1109/HPCA.2002.995703

12. DeVuyst M, Kumar R, Tullsen Dean M (2006) Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In: Proceedings International Parallel and Distributed Processing Symposium(IPDPS), pp 117–126. doi:10.1109/IPDPS.2006.1639374
13. Jiang Yunlian, Tian Kai, Shen Xipeng, Zhang Jinghe, Jie Chen, Tripath Rahul (2010) The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. IEEE Trans Paral Distrib Syst 22:1192–1205. doi:10.1109/TPDS.2010.193
14. Yunlian J, Xipeng S, Chen J, Rahul T (2008) Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: Proceedings of 17th Interantional Conference on Parallel Architectures and Compilation Techniques(PACT), pp 220–229. doi:10.1145/1454115.1454146
15. Zhuravlev S, Blagodurov S, Fedorova A (2010) Addressing shared resource contention in multicore processors via scheduling. In: Proceedings of Architectural support for programming languages and operating systems(ASPLOS), pp 129–142. doi:10.1145/1736020.1736036
16. Snavely A, Tullsen D (2000) Symbiotic job scheduling for a simultaneous multi threading processor. ASPLOS IX. doi:10.1145/356989.357011
17. Aamer J, Najaf-abadi Hashem H, Samantika S, Steely Simon C, Joel E (2012) CRUISE: cache replacement and utility-aware scheduling. ASPLOS XII 249–260 doi:10.1145/2150976.2151003
18. Gupta Saurabh, Xiang Ping, Zhang Yi, Zhou Huiyang (2013) Locality principle revisited: a probability-based quantitative approach. J Parallel Distrib Comput 73:1011–1027. doi:10.1016/j.jpdc.2013.01.010
19. Xiaoya X, Bao B, Ding C, Kai S (2012) Cache conscious task regrouping on multicore processors. In: Proceedings of 12th IEEE/ACM Interantional Symposium on Cluster, Cloud and Grid Computing. doi:10.1109/CCGrid.139
20. Knauerhase R, Brett P, Hohlt B, Li T, Hahn S (2008) Using OS observations to improve performance in multicore systems. IEEE Micro 28:54–66. doi:10.1109/MM.2008.48
21. Fedorova A, Seltzer M, Smith Michael D (2007) Improving performance isolation on chip multiprocessors via an operating system scheduler. In: Proceedings of 16th International Conference Parallel Architecture and Compilaton Techniques (PACT), pp 25-38. doi:10.1109/PACT.2007.40
22. Eiman E, Joo Lee C, Onur M, Patt Yale N (2010) Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. ASPLOS XV. doi:10.1145/1736020.1736058
23. Eklov D, Nikoleris N, Black-Schaffer D, Hagersten E (2011) Cache pirating: measuring the curse of the shared cache. In: International Conference on Parallel Processing (ICPP), pp 165–175. doi:10.1109/ICPP.2011.15
24. Perelman E, Polito M, Bouguet JY, Sampson J, Calder B, Dulong C (2006) Detecting phases in parallel applications on shared memory architectures. In: 20th IEEE Interantional Parallel and Distributed Processing Symposium (IPDPS), pp 88–98 doi:10.1109/IPDPS.2006.1639325
25. Han W, Xiaopeng G, Zhiqiang W, Yi L (2009) Using GPU to accelerate cache simulation. In: IEEE Interantional Symposium on Parallel and Distributed Processing with Applications, pp 565–570. doi:10.1109/ISP.2009.51
26. Curtin Ryan R, Cline James R, Slagle Neil P, March William B, Ram P, Mehta Nishant A, Gray Alexander G (2013) MLPACK: a scalable C++ machine learning library. J Mach Learn Res 801–805