Int J Parallel Prog (2015) 43:403–420
DOI 10.1007/s10766-013-0286-1

CCAP: A Cache Contention-Aware Virtual Machine Placement Approach for HPC Cloud

Hai Jin · Hanfeng Qin · Song Wu · Xuerong Guo

Received: 14 April 2013 / Accepted: 4 October 2013 / Published online: 22 October 2013

Abstract Applications in *High Performance Computing* (HPC) cloud are characterized by large cache resource consumption due to large-scale inputs and intensive communications, which creates serious *Shared Last Level cache* (SLLC) performance bottleneck. Current system software stacks are not efficient in addressing this issue among virtual machines at the hypervisor level or the threads at the operating system level. In this paper, we investigate performance interference due to contention for SLLC in the HPC cloud. We employ an enhanced reuse distance analysis technique with an accelerated cyclic compression algorithm to identify application's cache interference intensity. Based on reuse distance analysis, we propose a practical *Cache Contention-Aware virtual machine Placement* approach (CCAP). CCAP dispatches virtual machines according to their cache interference intensities to avoid cache pollution and interference, thus alleviating negative effects of cache contention. We implement CCAP in the Xen hypervisor. Evaluation of NPB workload reveals that CCAP can improve performance of cache sensitive applications when they are co-scheduled with cache pollution programs. For a 2-workload system, it reduces execution time by 12 %, as well as cache miss rate by 13 %, while increasing throughput by 13 %, on average. Moreover, CCAP also improves the average performance of the cache pollution programs by 5 %. For a 4-workload system, CCAP brings more significant performance improvement to cache sensitive applications, an average increase of 20 %.

Keywords HPC cloud · Cache contention · Reuse distance · Virtual machine placement

H. Jin (✉) · H. Qin · S. Wu · X. Guo
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology,
Wuhan 430074, China
e-mail: hjin@hust.edu.cn

1 Introduction

The *Shared Last Level Cache* (SLLC) is pervasively employed in today's *Chip Multi-Processor* (CMP) architecture [5]. By increased cache space and bandwidth utilization, SLLC plays an important role in improving system performance. However, SLLC also incurs cache pollution and interference problems due to contention for shared resource [4,25]. Interference with cache access results in more cache misses and degrades the performance of co-scheduled applications. Not only conventional processes in operating systems but also *Virtual Machines* (VMs) residing in a hypervisor [12,27,29] suffer from performance interference due to SLLC contention. As cloud computing has been the infrastructure of the modern computing ecosystem, increasing number of advanced academic and research institutes are migrating their applications to the *High Performance Computing* (HPC) cloud. Many of the virtual applications in the HPC cloud are characterized by large cache resource consumption due to their large-scale data inputs and intensive communications [1]. When multiple VMs holding virtual HPC applications are running separately on different physical cores, the problem of SLLC contention also arises and becomes even worse among the VMs that are located on cores belonging to the same chip. Cache contention breaks the inherent protection and isolation [6,30] provided by virtualization thus directly interferes performance of applications running within VM. It is no wondering that co-scheduling such HPC VMs causes significant performance degradation.

To study the performance interference caused by contention for SLLC quantitatively, we select workloads from the *NAS Parallel Benchmark* (NPB) suite [2] to perform various kinds of co-scheduling. Given its weak-locality and large cache working set, we deem CG as a cache pollution program and co-schedule it with other workloads. We compare the performance disparities between running the workload solo and co-scheduling them with CG. Figure 1 normalizes the observed performance disparities. Some workloads, such as LU and MG, present serious performance degradation with cache miss rates increased by approximately 15 %, execution time increased by more than 10 %, and throughput reduced by more than 11 %, respectively. In contrast, some workloads, including EP and SP, have less obvious performance degradation (less than 4 %).

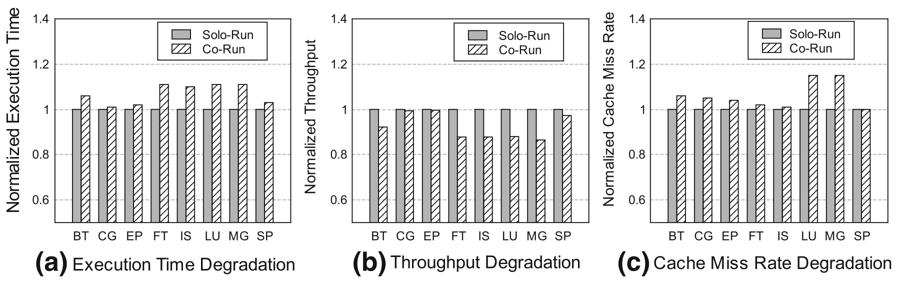

Fig. 1 Normalized performance degradation of NPB workloads when co-running with CG compared to running solo

Although SLLC contention is a serious problem for the HPC cloud, current system software stacks are not efficient in addressing this problem among virtual machines at the hypervisor level or threads at the operating system level. To alleviate performance degradation problem due to contention for SLLC, prior researchers proposed many solutions [15], including scheduling of threads [11,33] and partitioning cache [7, 21,26]. However, these approaches either need additional hardware support or have limitations in programming language or present high implementation complexities. Considering the effectiveness of commodity-virtualized systems, a practical software approach to address SLLC contention among applications in the HPC cloud that could be easily integrated into existing commodity-virtualized systems is required.

In this paper, we employ the reuse distance analysis technique to identify applications' cache interference intensity. Reuse distance analysis measures a program's memory reuse distance profile to characterize its cache behavior [10,23]. However, on-line reuse distance analysis is quite time consuming and sacrifices effectiveness. To reduce reuse distance complexity, we design an off-line cyclic compression algorithm to accelerate reuse distance computing. Based on this enhanced reuse distance analysis we classify HPC applications into three categories: cache pollution programs, cache sensitive programs, and cache friendly programs. Further, we propose a practical software solution, *Cache Contention-Aware virtual machine Placement* (CCAP), to address performance interference due to contention for SLLC in the HPC cloud. CCAP dispatches virtual machines according to their cache interference intensities. VMs running cache pollution applications are scheduled to cores that do not share cache with VMs running cache sensitive applications, therefore avoiding cache interference and pollution. We implement CCAP in an in-house virtual machine management kit based on the Xen [3] hypervisor. Evaluation of NPB shows that CCAP significantly improves the performance of cache sensitive applications when they are co-scheduled with cache pollution applications. For the 2-workload system, it reduces execution time by 12 % and the cache miss rate by 13 %, as well as increases throughput by 13 % on average. Moreover, for cache pollution applications, CCAP also has a 5 % performance improvement. For the 4-workload system, CCAP brings more significant performance improvement to cache sensitive applications (on average, more than 20 % improvement).

In summary, the main contributions of this paper are:

– We enhance conventional reuse distance analysis via a cyclic compression algorithm. The cyclic compression algorithm employs slice computing to accelerate reuse distance histogram computing. Compared with prior work, it decreases the spatial complexity due to large-scale memory accessing in data computing.
– We propose a classification for HPC cloud applications based on cache interference intensity. Cache interference intensity characterizes cache access behavior of the HPC cloud applications accurately. Evaluations show that such classification is especially useful for virtual machine placement to achieve performance isolation.
– We design a CCAP to address contention for SLLC. Based on cache interference intensities, CCAP captures applications' cache behavior, thus dispatching virtual machines to distinct cores and protecting cache sensitive applications from interference of cache pollution applications.

– We implement CCAP in an in-house virtual machine management kit based on the Xen hypervisor and carry detailed performance study on the popular NPB workload. CCAP is a practical software solution and does not need any extra hardware support. Evaluations reveal that by implicit partitioning SLLC among virtual machines, CCAP significantly improves performance.

The rest of the paper is organized as follows. Section 2 describes the enhanced reuse distance analysis technique with the cyclic compression algorithm to reduce reuse distance histogram complexity. Section 3 presents our application classification for the HPC cloud based on cache interference intensity. The main components of CCAP are also introduced in this section. Section 4 explains how to implement CCAP in a real system based on the Xen hypervisor. Section 5 reports our experimental methodology and performance evaluation on the NPB. Section 6 summarizes the work most closely related to ours. Section 7 concludes this paper and discusses our future work.

2 Enhanced Reuse Distance Analysis

2.1 Reuse Distance Analysis

Reuse Distance (RD) was originally called *Least Recently Used* (LRU) stack distance. RD is the number of distinct cache line accesses between two consecutive references to the same cache line. It is commonly used to characterize cache access behavior of applications through measuring the length of intervening data between two cache accesses. Further, we can construct a *Reuse Distance Histogram* by sorting all memory accesses based on their reuse distances. The reuse distance histogram indicates the distribution of reuse distances with different memory access percentages. We can exploit the reuse distance histogram to calculate cache hits and misses on different cache configurations and by distinct program input. Moreover, reuse distance histogram can be used to distinguish and classify different cache behaviors in shared last level cache (SLLC) as well. As an example, Fig. 2 presents the reuse distance histogram of programs from the NPB benchmark suite [2] with inputs of class S. The horizontal axis indicates different reuse distances (measured per kilobytes); the vertical axis records the corresponding distribution percentage of that reuse distance. The larger the percentage it occupies, more data a program will access at the corresponding reuse distance, which is another measurement of cache working set.

The inputs of reuse distance analysis are the applications' memory access address records. Different inputs may result in different memory access numbers, and thus determine the size of records directly. The size is usually very large. For example, even fed with inputs of class S, workloads from the NPB suite still have memory access address records in sizes of millions or even billions of bytes.

2.2 Cyclic Compression Algorithm

Considering the huge size of memory access address recording, it is impossible to compute reuse distance histograms online in a reasonable interval without any extra

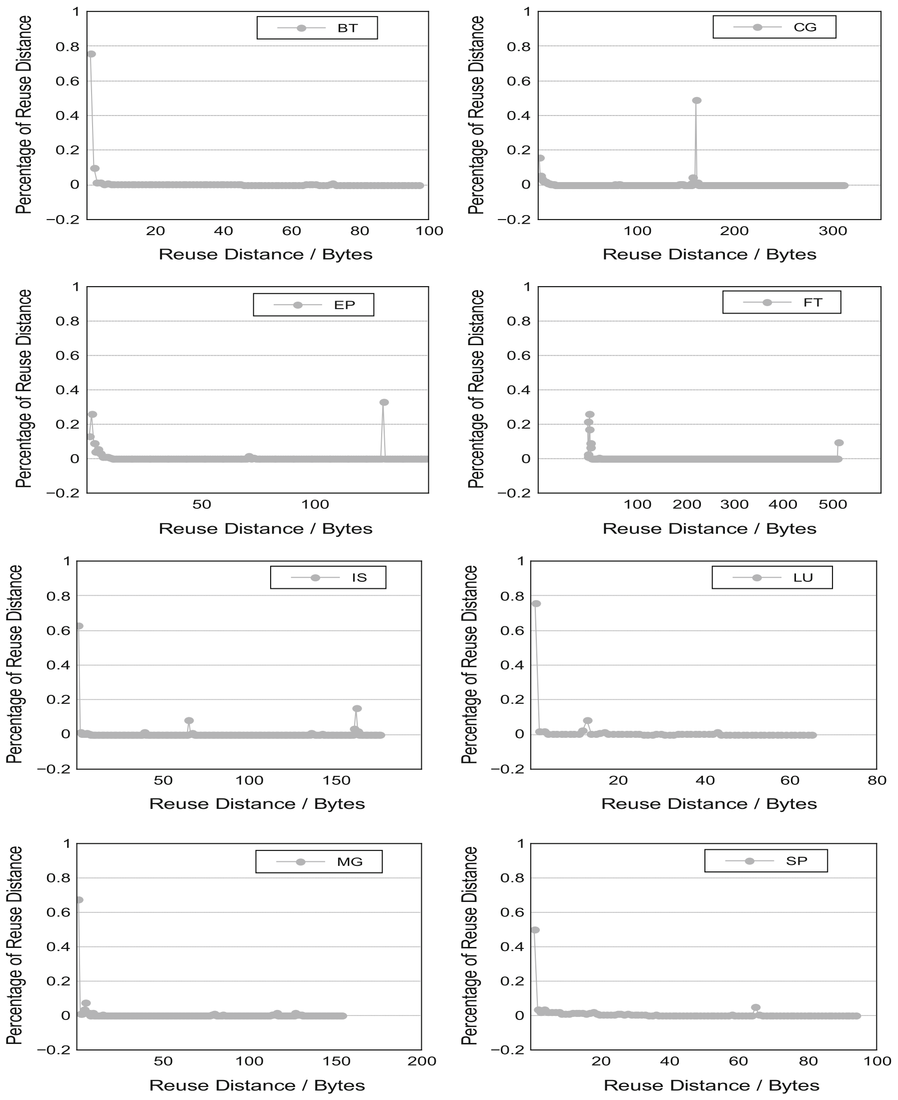

Fig. 2 Reuse distance histogram of applications from the NPB suite with input of class S

hardware support. Even conventional off-line computation is time consuming featured with both CPU and memory intensive. Given the large scale of data computing, we propose a cyclic compression algorithm to release memory store pressure and CPU processing loads. Based on the application's memory access address record size, this algorithm splits the record file into slices of equal size. Each slice is associated with a unique identity number and a specified slice length K_{min} (e.g., 10s thousand). Then, it forks multiple threads, which equals to available free CPU core counts, to generate the computing results of all file slices in parallel. The final computing result contains the reuse distance and a group of new memory access address records without any duplicate addresses. Finally, it reduces each file slice's computing result and summarizes the reuse distance results. The new memory access address records are also merged to a new memory access address record. The above operations are repeated through a loop until the most recent two numbers of file slices are equal. At this time, no duplicate memory accesses exist in this file slice length, which means that the next reuse distance to be computed is above the slice length. Thus, the file slice length needs to be upgraded for larger reuse distance calculation. Then, the algorithm increases file slice length, and starts with new cyclic processing until the number of file slices is one or reaches millions. The number that equals millions means that the reuse distance is more than 512K. At this time, it is enough to determine the application's reuse distance histogram. We choose 512K as a threshold because we learn from the reuse distance distribution fed with inputs of class S, as we will note in Sect. 5.2.

Algorithm 1 shows the detailed computation process of the reuse distance histogram enhanced by the cyclic compression algorithm described above. We denote the length of the applications' memory access address record as N, the maximum length of the file slice as K_{max}, and the number of threads as t. Thus, it is easy to conclude that the temporal and spatial complexity of this algorithm is $O(N^2)$ and $O(t * K_{max})$, respectively. Parameters of t and K_{max} can be tuned to optimize memory usage.

3 Cache Contention-Aware VM Placement

3.1 HPC Cloud Application Classification

The reuse distance histogram indicates the distribution of each individual reuse distance. Therefore, it is accurate and easy to distinguish different applications' cache behaviors when they contend for SLLC. We define *cache interference intensity* from the reuse distance histogram to indicate the length of reuse distance. Based on cache interference intensity, we propose a HPC cloud application classification to characterize cache access behavior in contending for SLLC. According to this classification, we cluster HPC cloud applications into three categories:

- Cache pollution applications, which refer to those applications that occupy large cache capacity and features with a relatively high percentage large reuse distance. Cache pollution applications waste cache resources and incur frequent cache line replacement, thus undermining cache hit rate and impairing performance of co-runners.
- Cache sensitive applications, which represent those applications that are strongly dependent on available cache resources. If the required cache capacity decreases, performance of cache sensitive applications will be degraded substantially.
- Cache friendly applications, which achieve good performance but consume relatively less cache capacity. Cache friendly applications do not take over extra cache capacity; therefore, it is safe to co-run cache friendly applications with other applications.

We perform reuse distance analysis enhanced with the cyclic compression algorithm on the NPB suite. Here, we only enforce inputs of class S to validate the effectiveness

Algorithm 1 The cyclic compression algorithm

```
Require: $length()$ returns the length of $f$, $threads.start()$ starts $thread$ to accomplish its tasks and
    $threading.join()$ waits for all threads in $threadlist$ to finish tasks.
1: $sliceLength \leftarrow K_{min}$
2: $accessAddressRec \leftarrow RawaccessAddressRec$
3: while $sliceLength$ lg $K_{max}$ do
4:    $recordLength \leftarrow length(accessAddressRec)$
5:    $sliceNumber \leftarrow recordLength/sliceLength$
6:    $flag \leftarrow True$
7:    while $flag$ do
8:       slice $accessAddress$, give each slice a identify number,
         produce $recordSlice[sliceNumber]$
9:       $threadNum \leftarrow memSpace/sliceLength$
10:      $tasksPerThr \leftarrow sliceNumber/(threadNum - 1)$
11:      for $numin[0..threadNum - 1]$ do
12:         tasks of calculate $recordSlice[num * TasksPerThr, (num - 1) * tasksPerThr - 1]$
            assign to $threadList[num]$
            $threadsList[num].start()$
13:      end for
14:      $threadList.join()$, $produceslice_R D[sliceNumber]$, $slice_A A[sliceNumber]$
15:      for $numin[0..sliceNumber - 1]$ do
16:         summarize $slice_R D[num]$ to $reuseDistanceRec$
            merge $slice_A A[num]$ to $NewaccessAddressRec$
17:      end for
18:      $accessAddressRec \leftarrow NewaccessAddressRec$
19:      $recordLength \leftarrow length(accessAddressRec)$
20:      $NewsliceNumber \leftarrow recordLength/sliceLength$
21:      if $NewsliceNumber = 1$ then
22:         calculate reuse distance of $accessAddressRec$,
            summarize result to $reuseDistanceRec$
            return $reuseDistance$
23:      end if
24:      if $NewsliceNumber = sliceNumber$ then
25:         $flag \leftarrow False$
26:      else
27:         $sliceNumber \leftarrow NewsliceNumber$
28:      end if
29:   end while
30:   $sliceLength \leftarrow sliceLength * 10$
31: end while
32: return $reuseDistanceRec$
```

of CCAP quickly. If time allows, relative large inputs can be fed to programs as well. Figure 2 lists the total reuse distance histograms for all programs.

Considering the impact of data input size on cache behavior, in our study, we can not guarantee programs in the NPB suite present the exact same reuse distance histograms as with inputs of class S when fed with ordinary inputs such as class A or class B. Such a case is possible, especially for cache sensitive applications. However, we can infer the estimated reuse distance distribution curve trends from small-scale behavior. We feed a cyclic compression algorithm with inputs of class W and class S individually and find that the accuracy of estimation of S from W is reasonable. Moreover, we enforce measuring of cache miss rates with inputs of all possible class not only class S and class W, thus preventing estimation from misclassification due to disparities between

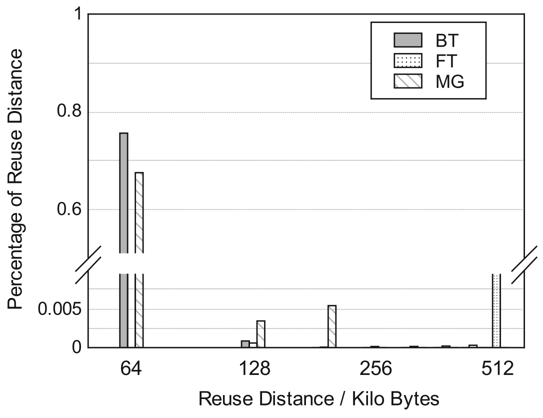

Fig. 3 Reuse distance histogram of programs from 3 categories. FT, MG, and BT are cache pollution, cache sensitive, and cache friendly applications, respectively

the reuse-distance histogram and various inputs. However, quantitative estimation of cache behaviors of large input from small input is beyond the scope of our study.

We correlate the cache miss rate and reuse distance distribution described below. When the size of the input data-set increases, the cache sensitive application has an obvious increase in cache miss rate. The increase of cache miss rate means that the percentage of several middle reuse distances substantially improved with the input data size. When using the ordinary input data size, these applications would suffer more cache misses from interference of co-schedulers. Cache friendly applications have small reuse distances, and more than 60 % of accesses falls in the low reuse distance block. As the input data size changes, these applications maintain stable low cache miss rate.

The typical reuse distance histograms of three categories are shown in Fig. 3. Based on our classification, CG and FT are cache pollution applications. Their reuse distances are as large as approximately 512 KB. LU and MG belong to cache sensitive applications. They have reuse distances of medium size and have large performance variations, even with little change of cache size. The remaining IS, BT, EP and SP share the same cache friendly application group. They all have small reuse distances within 100 KB.

3.2 Components of CCAP

To address contention for SLLC in the HPC cloud, we design a CCAP based on the above cache interference intensity classification from reuse distance analysis. CCAP computes the reuse distance histogram of each application from sampled memory access address records. It classifies applications into three categories based on reuse distance analysis and finally schedules virtual machine according to optimal placement solution. The purpose of CCAP is to reduce cache interference and minimize the impact of serious cache contention among different HPC cloud applications. As Fig. 4 shows, CCAP consists of two key processes, reuse distance analysis process and virtual machine placement process. Four modules are designed to implement the two processes. We describe these modules as follows.

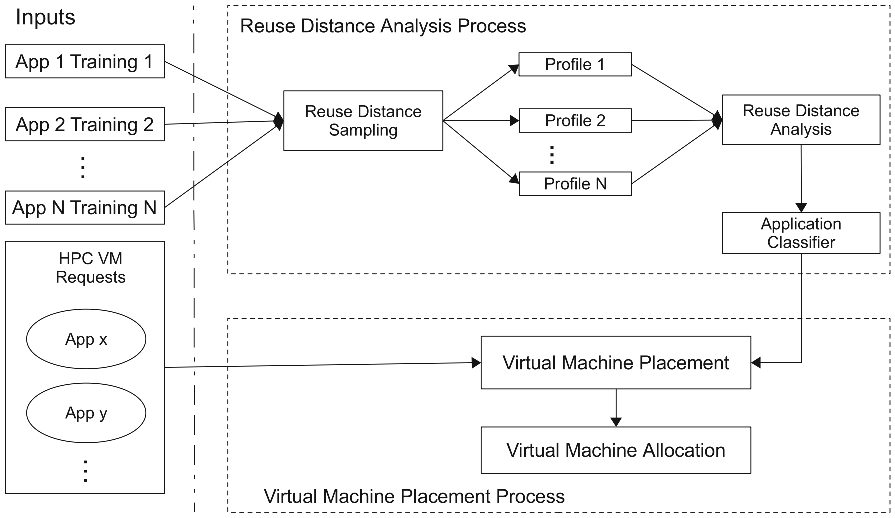

Fig. 4 The overview diagram of CCAP components. CCAP contains two key processes, reuse distance analysis and virtual machine placement. The rectangles inside each process describe the workflow

3.2.1 Reuse Distance Sampling Module

This module captures applications' reuse distance sampling. Two small test trainings in different sizes are fed to this module. The final sampled data consists of memory access addresses and cache misses.

3.2.2 Reuse Distance Analysis Module

This module plays a key role in the reuse distance analysis process. It computes applications' reuse distance histograms with the enhanced cyclic compression algorithm on the memory access address records provided by the reuse distance sampling module. Cache interference intensity is computed automatically from the generated reuse distance histograms and sampled cache miss rate, which directs application classification later.

3.2.3 Virtual Machine Placement Module

This module is responsible for dispatching the virtual machine to optimal cores. From application classification type, it responds to requests from the HPC cloud application by producing optimal virtual machine mapping to the cores. This mapping attempts to alleviate contention for SLLC to the minimum possible.

3.2.4 Virtual Machine Allocation Module

This module executes direction of the optimal virtual machine placement solution drawn from the virtual machine placement module. Through enforcing core binding to distinct fixed physical cores, it achieves virtual machine isolation, alleviating contention for SLLC.

3.3 Scheduling Virtual Machines in CCAP

We explain how CCAP schedules virtual machines in details in this subsection. CCAP relies on virtual machine placement module to select VMs and virtual machine allocation module to allocate virtual processors to the corresponding VMs.

CCAP selects virtual machines according to their scheduling priorities which are correlated with their cache interference intensities. From the classification in reuse distance analysis, we observe that different categories have different impacts on contention for SLLC. The cache pollution applications affect co-runners seriously. The cache sensitive applications have a strong dependency on cache resources, but they have little impact on others. The cache friendly applications are always friendly to others. Based on this observation, we assign virtual machines of different categories with corresponding scheduling priorities. The cache pollution applications have the highest priorities, the cache sensitive applications have the medium priorities, and the cache friendly applications have the lowest priorities. The virtual machine placement module schedules VMs from high priority to low priority. It picks the cache pollution applications first. Then, it locates cache sensitive applications and finally handles cache friendly applications.

On the other hand, to minimize the entire system's cache contention, CCAP allocates virtual processors in the following sequences. For the cache pollution applications, the empty processors come first, followed by processors that already have cache friendly programs located, processors that already have cache pollution applications loaded are utilized subsequently, followed by processors that co-run cache sensitive applications. This allocation sequence guarantees the minimization of the probability that both cache pollution co-runners would be affected. Since this category affects others mostly in cache contention, we prefer to reduce its cache interference to others. In contrast, for the cache sensitive category, the virtual processor allocation sequence is empty, the cache friendly category, the cache sensitive category and finally the cache pollution category. For the cache friendly category, the virtual processor allocation sequence is empty, the cache friendly category, the cache pollution category, and the cache sensitive category follows.

4 System Implementation

We implement CCAP in an in-house virtual machine management kit based on the Xen [3] hypervisor. This kit consists of one central management node and a group of computing data nodes, both of which are running on physically independent machines. Virtual machines are scheduled to run on the data nodes. The collected total statistics are stored in the central node, which also controls the entire data nodes as a cluster administration management node. We integrate CCAP into our prototype system, as Fig. 5 demonstrates. As a high level logic control component, the virtual machine placement module of CCAP is mainly implemented in the central node to take advantage of the

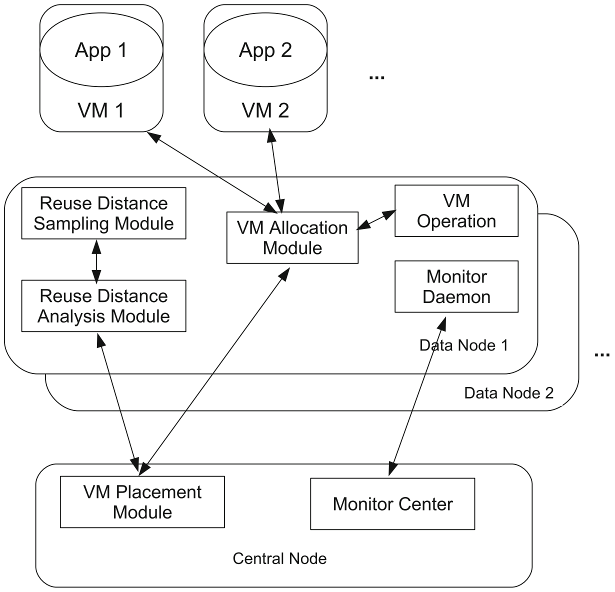

Fig. 5 Architecture of the CCAP implementation

stored information to determine the optimal virtual machine schedule solution. The remaining components of CCAP are deployed on the data nodes. Applications' reuse distance files produced from the reuse distance sampling module are analyzed by the reuse distance analysis module, which generates application classification based on cache interference intensity. Based on the classification result received, the central node dictates the virtual machine allocation module listening to the data nodes to schedule the virtual machine to the corresponding optimal designated cores according to the determined mapping solution. Core binding is enforced to prevent unexpected dynamic virtual machine migration from the inherent scheduling algorithm of the Xen hypervisor until the newly mapping solutions are produced.

The central node is responsible for handling multiple virtual machine requests, and generating virtual machine placement solutions according to the applications' classification. It collects the CPU usage information and virtual machine running statistics through monitor center component, which communicates with monitor daemon components in each data node. The information combined with classification is used to compute the optimal virtual machine placement solution. The virtual machine module repacks requests from virtual machines with physical core allocation information, which is then sent to the virtual machine allocation module on the data nodes. It also receives feedback information from the data nodes.

Each data node is the platform to load the virtual machines. The data node executes specific virtual machine commands sent from the central node. It also gains application classification through the reuse distance sampling module and reuse distance analysis module, which is transmitted to the central node. When it receives the virtual machine placement solution from the central node, it allocates virtual machine by invoking the virtual machine allocation module. In addition, the monitor daemon on the data node collects the virtual machine statistics and CPU usage information, which are reported to the monitoring center.

5 Experiment and Evaluation

5.1 Experimental Setup

We implement CCAP based on Xen 4.0.1. Our physical platform consists of a central node and a data node, as presented in Sect. 4. The data node is equipped with the Xen hypervisor. The physical platform is equipped with Intel Xeon E5310 and 24 GB main memory. It contains 2 quad-core processors. The cores in each processor share a 4 MB, 16-way set associative L2 cache (SLLC) with a 64 bytes cache line. Each core has a private 32 KB L1 instruction cache and 32 KB L1 data cache. Eight programs from NPB 3.3, including BT, CG, EP, FT, IS, LU, MG, and SP, are selected to construct our HPC cloud workloads.

5.2 Methodology

We adopt three performance metrics: *execution time*, *throughput*, and *cache miss rate*. Specifically, we obtain execution time and throughput from the final statistics outputs of the NPB suite. We also employ XenOprofile [8], a commonly used performance counter profiling toolkit in the Xen virtualized platform, to collect cache miss rate. Since XenOprofile does not obtain cache miss rate directly, we sample the total number of LLC misses (NR_LLC_MISSES) and the total number of LLC references (NR_LLC_REFS) to compute the cache miss rate $\left(\frac{NR\_LLC\_MISSES}{NR\_LLC\_REFS}\right)$. The average sampling interval is configured to 1000 cycles.

Considering the time-consuming run of a huge input, we feed the reuse distance sampling module for each application from the NPB suite with the training inputs of class S. We use a customized PinTools [22] to record applications' memory access address. After several manual experimental measurements, we finally manage to find that the size of memory accesses record files ranges from 74 MB to 2.1 GB, as well as the maximum reuse distance is approximately 512 KB. Based on this observation, we configure the main parameters of the cyclic compression algorithm as follows. The thread number is set to 8, the available free core counts of our platform. The maximum slices number is set to 1 million, which is large enough to collect reuse distances of less than 512 KB.

Our HPC cloud classification based on cache interference intensity is enforced to construct workload. Unlike reuse distance sampling, we feed workloads with a large input size of class A. Since cache pollution is highly correlated with other co-running applications sharing the same physical node, we cover all possible combination pairs to evaluate the effectiveness of CCAP. Specially, we consider pairs towards both a 2-workload system and a 4-workload system. Through our control switch at the central node, we determine whether to activate the CCAP component. Compared pair runs are

Table 1 Performance comparison of FT and MG when co-scheduled with the default configuration and CCAP, respectively

Workload	Default		CCAP	
	FT	MG	FT	MG
Time (s)	50.79	25.76	43.90	22.13
Throughput (Mop/s)	137.82	759.93	162.45	877.97
L2 cache miss rate (%)	48.14	62.76	45.64	55.45

conducted, one utilizing CCAP and the other without. Average performance metrics are collected for five runs.

5.3 Performance Evaluation

5.3.1 Spatial Complexity of Cyclic Compress Algorithm

Above all, we discuss the spatial complexity of the cyclic compression algorithm. As noted in Sect. 2.2, for N bytes memory access records, the conventional reuse distance analysis procedure is featured with spatial complexity of $O(N)$. In contrast, our cyclic compression algorithm consumes space of only $O(t * K_{max})$, where t and K_{max} denote computing number and slice file size, respectively. Configured with the parameters mentioned in the above methodology, for an average 8 bytes of memory access records, the average memory consumption is reduced from 2.1 GB to 62 MB (8 $\times$ 1,000,000 $\times$ 8 bytes). It alleviates the memory pressure to the hypervisor significantly, servicing more memory requests from the upper virtual machines.

5.3.2 Performance of 2-Workload System

For a 2-workload system, we are mostly concerned with workloads consisting of a cache pollution program and a cache sensitive program. Therefore, we study how CCAP improves performance of a cache sensitive application when it is co-scheduled with a cache pollution program. According to our classification based on cache interference intensity, we select MG from the cache sensitive category and FT from the cache pollution group and co-schedule them. Performance statistics for the default run without any shared cache control (default) and the run with CCAP are presented in Table 1. The normalized performance improvements are also presented in Fig. 6. With the shared cache control from CCAP, MG's performance is improved, which reduces execution time by 14.09 % and cache miss rate by 11.66 %, while increasing throughput by 15.53 %. We also find that CCAP could improve FT (a cache pollution application) as well, via reducing execution time by 13.57 % and cache miss rate by 5.19 %, while increasing throughput by 17.15 %.

Beyond co-scheduling of FT and MG, we also explore performance for other co-scheduled programs. We find that for the combination of a cache sensitive program and a cache pollution one, CCAP has a more observable performance improvement. Figure 7 presents the typical performance improvement for applications matching such

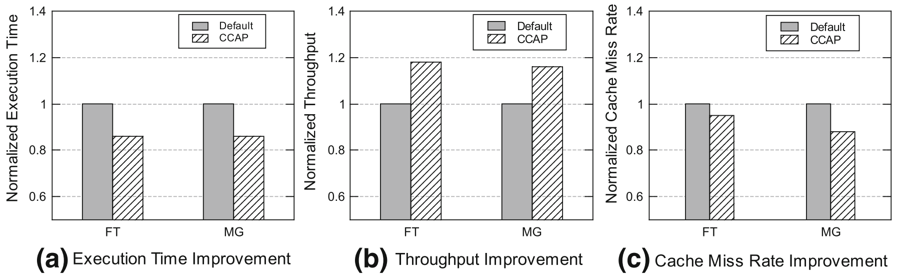

Fig. 6 Normalized performance improvement when co-schedule a cache sensitive application MG and a cache pollution program FT with the shared cache control from CCAP

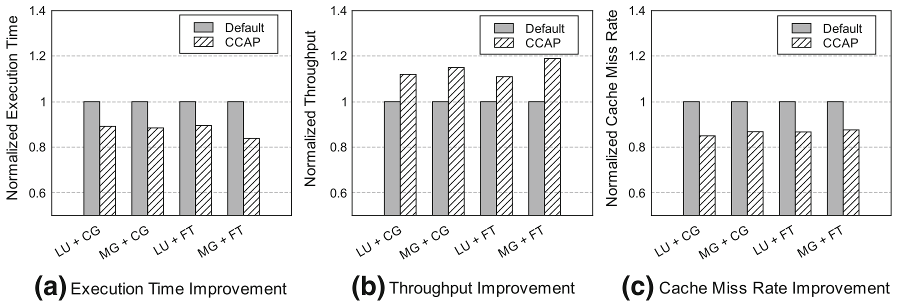

Fig. 7 Normalized performance improvement of typical cache sensitive applications (LU, MG) when co-scheduled with cache pollution applications (CG, FT) with shared cache control from CCAP.

a combination case. On average, it reduces execution time by 12 % and cache miss rate by 13 %, while increases throughput by 13 %. Our approach prevents cache sensitive applications from suffering serious performance degradation due to contention from cache pollution applications.

Besides cache sensitive applications, we also find that CCAP has unexpected performance improvement for cache pollution applications. For example, Fig. 8 shows the performance improvement of FT when it is co-scheduled with other workloads. On average, it reduces execution time by 11 % and cache miss rate by 5 %, while increasing throughput by 12 %.

5.3.3 Performance of 4-Workload System

In the scenario of 4-workload system, we study how the number of VMs per node affects the performance of applications sharing the same physical node. We measure several combined pairs where workloads consist of programs classified in three different categories: 2 cache pollution programs plus 1 cache sensitive program plus 1 cache friendly program (CG, CG, MG, and EP), 1 cache pollution program plus 2 cache sensitive programs plus 1 cache friendly program (CG, MG, LU, and EP), and 1 cache pollution program plus 1 cache sensitive plus 2 cache friendly programs (CG, MG, BT, and EP). We do not include the performance statistics of cache pollution applications and cache friendly applications due to page limits. We show the performance results of

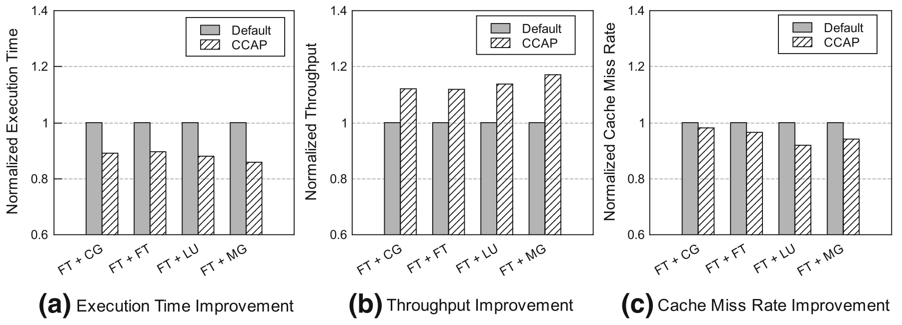

Fig. 8 Normalized performance improvement of FT (a cache pollution application) when it is co-scheduled with cache sensitive applications with shared cache control from CCAP

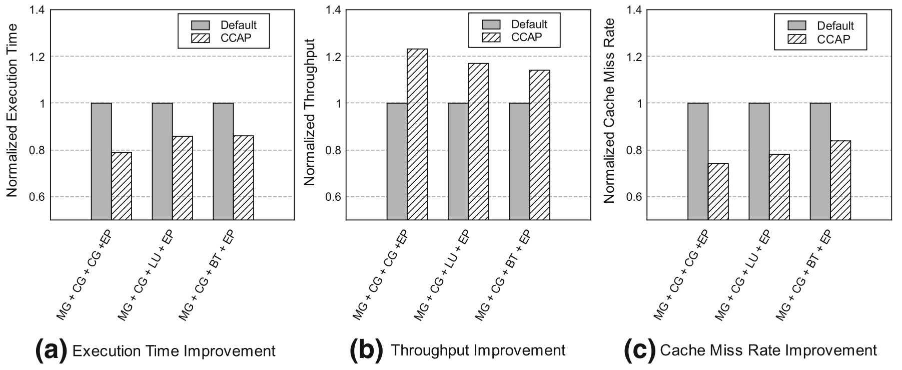

Fig. 9 Normalized performance improvement of typical cache sensitive workloads on 4-workload system with shared cache control form CCAP

the typical cache sensitive application (MG) in Fig. 9. The cache sensitive application achieves performance improvement, including reducing execution time by 21 % and cache miss rate by 23 %, while increasing throughput by 25 % (at maximum). Cache pollution is highly correlated with co-running applications. According to this study, we find that CCAP achieves greater performance improvement for this case, compared to the 2-workloads system.

6 Related Work

Prior researches have proposed solution to control and alleviate SLLC contention among threads in operating systems. However, current system software stacks are weak in addressing this problem for HPC cloud either at the hypervisor level or at the operating system level. In this section, we summarize the work most closely related to ours.

Prior cache resource management solutions aim for different goals, including system performance improvement [14,31], QoS ensurance [13,17,24] or cache usage fairness [18,20]. Several studies focus on analysis of the reuse distance histogram, which determines program cache access behavior [9,16,28,32]. These approaches adopt mechanisms such as scheduling resource, partitioning cache to realize cache allocation and isolation among threads or programs. The reuse distance analysis is used to measure cache interference, cache misses, or other cache characteristics. The multiple reuse distance analysis approaches make efforts to migrate cache contention or improve cache hits, which lead to programs' performance improvements [4,19].

The pain classification scheme [15] uses stack distance analysis to define cache sensitivity and cache intensity of programs. Cache sensitivity refers to how much impact a program suffers in cache contention with co-runners. Cache intensity refers to how much a program affects others in cache contention. The stack distance is used to calculate the two cache interference factors that are combined to compute the co-schedule "pain". Based on the pain classification scheme, the scheduler makes decisions on scheduling of threads. This approach is effective in reducing cache interferences among co-scheduled threads. However, it has a high implementation complexity due to cache interference factors computing, making it difficult to integrate into existing systems.

The soft-OLP approach [21] realizes cache partition in data object granularity based on locality pattern analysis. This approach recognizes each data object's locality pattern based on its reuse distance histogram and inter-object interference histograms. It samples smaller test training inputs to improve sampling efficiency. This approach uses the page coloring software method to partition cache among data objects of one program. The soft-OLP approach effectively reduces programs' cache misses and execution time. However, it is limited on programming languages and programming modes. Additionally, to collect data objects, it requires pre-scanning of a program binary file symbol table.

7 Conclusion and Future Work

Contention for SLLC incurs serious performance interference for HPC cloud applications when they are co-scheduled with others. In this paper, we investigate this performance interference from SLLC contention. We employ commonly used reuse distance analysis to characterize HPC cloud applications. The conventional reuse distance analysis technique is enhanced via an accelerated cyclic compression algorithm to reduce the spatial complexity of computing reuse distance histograms. Further, we classify the HPC cloud applications based on cache interference intensity and propose a CCAP. CCAP dispatches virtual machines according to cache interference intensity to prevent cache pollution. Finally, we implement CCAP in an in-house virtual machine kit based on the Xen hypervisor. Evaluation of the NPB shows that CCAP undermines SLLC contention intensity, thus significantly improving both performance of cache sensitive applications and cache pollution programs when they are co-scheduled.

In this paper, we construct reuse distance histograms via off-line analysis. To capture program run time phase changes precisely, a dynamic on-line and lightweight approach may be more suitable for complicated HPC applications. Our cache contention-aware classification scheme is not limited in virtual machine placement. We can further explore an advanced cache capacity management mechanism at the cache block level for the HPC cloud based on this scheme.

Acknowledgments We would like to thank the anonymous reviewers for their helpful comments. The research is supported by National Science Foundation of China under Grant No. 61232008 and 61073024, ChinaGrid and CRANE project, Outstanding Youth Foundation of Hubei Province under Grant No. 2011CD-A086, National 863 Hi-Tech Research and Development Program under Grant No. 2013AA01A213, and Research Fund for the Doctoral Program of MOE under Grant 20110142130005.

References

1. Alarm, S., Barrett, R.F., Kuehn, J.A., Roth, P.C., Vetter, J.S.: Characterization of scientific workloads on systems with multi-core processors. In: Proceedings of IEEE International Symposium on Workload Characterization (IISWC'06), pp. 225–236. IEEE (2006)
2. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., et al.: The nas parallel benchmarks—summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC'91), pp. 158–165. ACM (1991)
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03), pp. 164–177. ACM (2003)
4. Barker, D.P.: Realities of multi-core CPU chips and memory contention. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'2009), pp. 446–453. IEEE (2009)
5. Borkar, S.: Thousand core chips: a technology perspective. In: Proceedings of the 44th Annual Design Automation Conference, pp. 746–749. ACM (2007)
6. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multi-processor architecture. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05), pp. 340–351. IEEE (2005)
7. Chang, J., Sohi, G.S.: Cooperative cache partitioning for chip multiprocessors. In: Proceedings of the 21st Annual International Conference on Supercomputing (SC'07), pp. 242–252. ACM (2007)
8. Cohen, W.E.: Tuning programs with oprofile. Wide Open Mag. **1**, 53–62 (2004)
9. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pp. 245–257. ACM (2003)
10. Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., Veidenbaum, A.V.: Improving cache management policies using dynamic reuse distances. In: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12), pp. 389–400. IEEE (2012)
11. Fedorova, A., Seltzer, M., Smith, M.D.: Improving performance isolation on chip multiprocessors via an operating system scheduler. In: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07), pp. 25–38. IEEE (2007)
12. Goldberg, R.P.: Survey of virtual machine research. Computer **7**(6), 34–45 (1974)
13. Guo, F., Kannan, H., Zhao, L., Illikkal, R., Iyer, R., Newell, D., Solihin, Y., Kozyrakis, C.: From chaos to QoS: case studies in CMP resource management. ACM SIGARCH Comput. Archit. News **35**(1), 21–30 (2007)
14. Hao, S., Du, Z., Bader, D.A., Ye, Y.: A partition-merge based cache-conscious parallel sorting algorithm for CMP with shared cache. In: Proceedings of the 38th International Conference on Parallel Processing (ICPP'09), pp. 396–403. IEEE (2009)
15. Hsu, L.R., Reinhardt, S.K., Iyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06), pp. 13–22. ACM (2006)
16. Hsu, W.C., Chen, H., Yew, P.C., Chen, D.Y.: On the predictability of program behavior using different input data sets. In: Proceedings of 6th Annual Workshop on Interaction Between Compilers and Computer Architectures, pp. 45–53. IEEE (2002)
17. Iyer, R.: CQoS: a framework for enabling Qos in shared caches of CMP platforms. In: Proceedings of the 18th Annual International Conference on Supercomputing (SC'04), pp. 257–266. ACM (2004)
18. Jahre, M., Natvig, L.: A light-weight fairness mechanism for chip multiprocessor memory systems. In: Proceedings of the 6th ACM Conference on Computing Frontiers (CF'09), pp. 1–10. ACM (2009)

19. Jaleel, A., Theobald, K.B., Steely, S.C. Jr., Emer, J.: High performance cache replacement using re-reference interval prediction (rrip). In: Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10), pp. 60–71. ACM (2010)
20. Kim, S., Chandra, D., Solihin, Y.: Fair cache sharing and partitioning in a chip multiprocessor architecture. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04), pp. 111–122. IEEE (2004)
21. Lu, Q., Lin, J., Ding, X., Zhang, Z., Zhang, X., Sadayappan, P.: 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'09), pp. 246–257. IEEE (2009)
22. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), pp. 190–200. ACM (2005)
23. Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 78–117 (1970)
24. Nesbit, K.J., Laudon, J., Smith, J.E.: Virtual private caches. In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07), pp. 57–68. ACM (2007)
25. Nesbit, K.J., Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M., Smith, J.E.: Multicore resource management. IEEE Micro **28**(3), 6–16 (2008)
26. Qureshi, M.K., Patt, Y.N.: Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), pp. 423–432. IEEE (2006)
27. Rosenblum, M., Garfinkel, T.: Virtual machine monitors: current technology and future trends. Computer **38**(5), 39–47 (2005)
28. Schuff, D.L., Parsons, B.S., Pai, V.S.: Multicore-aware reuse distance analysis. In: Proceedings of 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Ph.d. Forum (IPDPSW'10), pp. 1–8. IEEE (2010)
29. Smith, J.E., Nair, R.: The architecture of virtual machines. Computer **38**(5), 32–38 (2005)
30. Soares, L., Tam, D., Stumm, M.: Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08), pp. 258–269. IEEE (2008)
31. Suo, G., Yang, X., Liu, G., Wu, J., Zeng, K., Zhang, B., Lin, Y.: IPC-based cache partitioning: an IPC-oriented dynamic shared cache partitioning mechanism. In: Proceedings of the 2008 3rd International Conference on Convergence and Hybrid Information Technology (ICHIT'08), pp. 399–406. IEEE (2008)
32. Zhong, Y., Dropsho, S.G., Shen, X., Studer, A., Ding, C.: Miss rate prediction across program inputs and cache configurations. IEEE Trans. Comput. **56**(3), 328–343 (2007)
33. Zhuravlev, S., Blagodurov, S., Fedorova, A.: Addressing shared resource contention in multicore processors via scheduling. In: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10), pp. 129–142. ACM (2010)