Cache Contention Aware Virtual Machine Placement and Migration in Cloud Datacenters

Liuhua Chen, Haiying Shen and Stephen Platt
Clemson University, Clemson, South Carolina 29634
Email: {liuhuac, shenh, splatt}@clemson.edu

Abstract—**In cloud datacenters, multiple Virtual Machines (VMs) are co-located in a Physical Machine (PM) to serve different applications. Prior VM consolidation methods for cloud datacenters schedule VMs mainly based on resource (CPU and memory) constraints in PMs but neglect serious shared Last Level cache contention between VMs. This may cause severe VM performance degradation due to cache thrashing and starvation for VMs. Current cache contention aware VM consolidation strategies either estimate cache contention by coarse VM classification for each individual VM without considering co-location and (or) require the aid of hardware to monitor the online miss rate of each VM. Therefore, these strategies are insufficiently accurate and (or) difficult to adopt for VM consolidation in clouds. In this paper, we formalize the problem of cache contention aware VM placement and migration in cloud datacenters using integer linear programming. We then propose a cache contention aware VM placement and migration algorithm (CacheVM). It estimates the total cache contention degree of co-locating a given VM with a group of VMs in a PM based on the cache stack distance profiles of the VMs. Then, it places the VM to the PM with the minimum cache contention degree and chooses the VM from a PM that generates the maximum cache contention degree to migrate out. We implemented CacheVM and its comparison methods on a supercomputing cluster. Trace-driven simulation and real-testbed experiments show that CacheVM outperforms other methods in terms of the number of cache misses, execution time and throughput.**

I. Introduction

Many cloud systems (such as those in Amazon [1], Google [2] and Microsoft [3]) provide IaaS service in the form of VMs. They consolidate multiple VMs on the same physical machine (PM) in order to optimize the underlying resource usage of the cloud datacenters when providing service to different customers. In spite of the high resource utilization due to resource sharing, the resource contention between VMs often lead to significant performance degradation of VMs. After the initial VM allocation to PMs, VM migration methods [4] are usually used to migrate VMs from overloaded PMs to underloaded PMs to avoid resource contention. Efficient and effective VM allocation and migration became one of the major challenges that datacenter operators face in order to satisfy Service Level Agreements (SLAs) [5] between a cloud customer and the cloud service provider, especially when the number of VMs increases rapidly nowadays in clouds.

An effective VM allocation algorithm should allocate as many VMs as possible to a PM while i) meeting their explicit requirements on resources such as CPU and memory and ii) minimizing contentions on Last Level cache (LLC). Many previous VM allocation or migration methods provide a metric to choose destination PM and migration VM to handle objective i) but neglect objective ii). Current VM placement and migration approaches mainly consider resource (e.g., CPU and memory) constraints in PMs. They estimate VM performance based on either measured [6]–[10] or predicted [11]–[13] VM resource utilizations, but neglect resource utilization overhead [14] due to virtualization such at serious shared LLC contention between VMs, which may seriously degrade VM performance. Cache contention refers to the situation that a VM suffers extra cache misses because other co-located VMs on the same processor bring their own data into the LLC, which evicts the VM's data, and then the VM brings its own data into the LLC which causes cache misses to other VMs. Cache contention breaks the protection and isolation [15] provided by virtualization, thus directly degrading the performance of VMs. Many applications running on clouds are characterized by large cache resource consumption and various cache access behaviors due to their large-scale data inputs and intensive communications (e.g., between the application and end users and between front-end servers and back-end databases of the application) [16]. When multiple VMs that host these applications are co-located, the problem of LLC contention becomes even worse.

Some cache contention aware VM consolidation strategies [17]–[20] in the high performance computing environment have been proposed. However, they estimate cache contention by coarse VM classification for individual VM without considering co-location. First, they conduct VM classification based on rough estimation of cache contention (e.g., cache pollution, cache sensitive and cache friendly) and avoid co-locating VMs that will generate many cache misses (e.g., cache pollution VMs and cache sensitive VMs). Then, they are not suitable for the practical cloud environment, which hosts numerous VMs with various cache access characteristics. Second, these strategies estimate the cache contention for individual VMs without the effect of co-location, that is, they do not measure how much the VM will suffer or other VMs will suffer when the VM is co-located with other VMs. Third, they only aim to isolate cache pollution VMs from cache sensitive VMs, but fail to evaluate the cache contention for every possible location for a VM to find the location with the minimum performance degradation.

In this paper, we formalize the problem of cache contention aware virtual machine placement and migration in clouds using integer linear programming. We then propose a heuristic Cache Contention Aware Virtual Machine Placement and Migration algorithm (CacheVM). It estimates the total cache contention degree of co-locating a given VM with a group of VMs in a PM based on measured cache stack distance profiles of the VMs. We claim that these profiles are efficient for predicting VM runtime caching behavior under complex co-location scenarios for two reasons. First, a large number of jobs in a commercial cloud recur with predictable resource requirements [21]. Second, these profiles are obtained when the jobs are running with full resource provisioning rather than in a resource constrained environment, where the monitored VM co-locates with other VMs. Therefore, these profiles are more accurate in predicting VM caching behaviors. Then, CacheVM places the VM to the PM with the minimum cache contention and chooses the VM from a PM that generates the maximum cache contention to migrate out. Though CacheVM focuses on reducing cache contention, it can be easily combined with other VM placement schemes [9], [22] to jointly consider both cache contention and resource utilization (e.g., CPU and memory). We used benchmarks and implemented CacheVM in a supercomputing cluster. Trace-driven simulation and real-world testbed experiments show that CacheVM significantly reduces average performance degradation experienced by the VMs compared to other methods. The contributions of this paper are summarized as follows:

- We propose a cache contention aware VM performance degradation prediction algorithm (CCAP) for estimating the stack distance profiles of VMs.
- We present an optimization formulation for a cache contention aware VM placement problem.
- We propose a cache contention aware VM placement and migration algorithm (CacheVM) based on CCAP.
- We carry out trace-driven simulation and prototype CacheVM on a supercomputing cluster.

The rest of this paper is organized as follows. Section II briefly describes the related work. Section III introduces the background of cache interference in shared LLC. Section IV presents the performance degradation prediction method. Section V presents the design of CacheVM. Section VI presents the performance evaluation of CacheVM. Finally, section VII summarizes the paper with remarks on our future work.

II. Related Work

Current VM placement and migration approaches for cloud mainly consider CPU and memory resource constraints in PMs. Chen *et al.* [6] proposed a resource intensity aware load balancing method to achieve load balance in cloud datacenters. The authors also proposed a scheme that predicts VM resource utilization and consolidates VMs with complementary utilization [9]. However, these methods mainly consider CPU and memory resource constraints in PMs, but neglect the cache contention between co-located VMs, which can also significantly degrade VM performance.

Some cache contention aware VM consolidation strategies have been proposed for the high performance computing environment. Xie *et al.* [17] classified the applications into four different classes based on their miss rate and provided estimates of how well various classes co-exist with each other on the same shared cache. Jin *et al.* [18] applied a similar classification method to categorize VMs that aims to relieve cache contention in VM placement. However, these methods roughly estimate cache contention by coarse VM classification, so they are insufficiently accurate in the practical cloud environment, which hosts numerous VMs with various cache consumption characteristics, because the small number (three or four) classes are far from sufficient in capturing the performance of a huge amount of VMs from different applications. Knauerhase *et al.* [19] proposed MissRate algorithm that estimates the miss rates of VMs online, and then allocates a VM to the PM that has the least sum of the miss rates of its hosted VMs. Kim *et al.* [20] proposed a VM consolidation algorithm based on their cache interference model. It calculates interference-intensity and interference-sensitivity (based on miss rate and miss ratio) to classify VMs and tries to co-locate highly interference-intensive VMs and less interference-sensitive VMs. However, these two methods only consider the metrics of individual VMs and hence are not sufficiently accurate to reflect VM performance in a complex VM co-location environment. As we indicated in Section I, these methods are not suitable for the cloud environment. Some other methods [23], [24] focus on online monitoring and leverage live VM migration to reduce contentions on the cache. Ahn *et al.* [23] proposed and evaluated online VM scheduling techniques for cache sharing based on LLC misses in hardware performance monitoring couters. Wang *et al.* [24] proposed an architecture aware scheme to take into account cache interference when making migration decisions. However, these methods rely on hardware counters to periodically keep track of per-VM LLC misses, which is not feasible in many cloud vendors.

III. Background

We first present a brief review of cache hierarchy, especially the process to share LLC among CPU cores. Today's multi-core processors have a hierarchy of caches, typically one or more caches private to each core and a single LLC shared across all cores. A VM's memory is mapped to the physical memory and data is inserted into and evicted from the cache hierarchy at the granularity of a cache line (64 bytes for most processors).

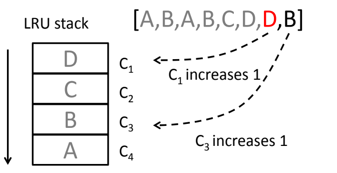
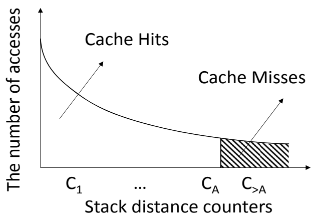
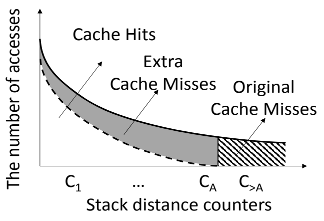


Fig. 1: LRU stack and counters in stack distance profile.

A stack distance profile (also referred to as marginal gain counters [25]) captures the temporal reuse behavior of an application in a fully- or set-associative cache. It is obtained by monitoring cache accesses on a system with a Least Recently Used (LRU) cache. When a cache line is accessed, the line is pushed (or moved) to the top of the LRU stack



(a) Stack distance profile



(b) Extra cache misses in cache contention

Fig. 2: Stack distance profiles.

of a set. As shown in Figure 1, for an A-way associative cache, a stack distance profile of VM i is represented by $f_i = \{C_1, C_2, ..., C_A, C_{>A}\}$, where d in C_d corresponds to each LRU stack position, C_d counts the number of hits to the line in the d^{th} LRU stack position and $C_{>A}$ counts the number of cache misses. Note that the first line in the stack is the most recently used line in the set, and the last line in the stack is the least recently used line in the set. If it is a cache access to a cache line in the d^{th} position in the LRU stack of the set, C_d is incremented. If it is a cache miss, i.e., the line is not found in the LRU stack, then the miss counter $C_{>A}$ is incremented.

Figure 2(a) shows an example of a stack distance profile. Applications with temporal reuse behavior usually access more-recently-used data more frequently than less-recently-used data. Therefore, typically, the stack distance profile shows decreasing values from the left to the right. The number of cache misses for a smaller cache can be easily computed using the stack distance profile. For example, for a smaller cache that has A' ($A' < A$) associativity, the new number of misses can be computed as $C_{>A} + \sum_{d=A'+1}^{A} C_d$.

In order to predict the performance of VMs that are affected by VM co-locations, it is necessary to compare the stack distance profiles from different VMs. Because the number of cache accesses within a unit time collected in different processors may differ greatly due to their different computing capabilities, we need to divide each of the counters by the number of processor cycles in which the profile is collected. That is, $C_i = \frac{C_i}{CPU_{cycle}}$ and $C_{>A} = \frac{C_{>A}}{CPU_{cycle}}$. Then, we define the miss frequency as $C_{>A}$ and define the reuse frequency as $\sum_{i=1}^{A} C_i$. The sum of these two frequencies is referred to as access frequency.

IV. Degradation Estimation

A. Cache Contention Prediction

Figure 2(b) shows an example of the transformation of a stack distance profile due to cache contention. When a VM is co-located with some other VM, some of its original hits will turn into misses because of the data eviction by the other VMs. The original hits that turn into misses due to VM co-location are referred to as *extra cache misses*, as indicated by the grey area. The value of extra misses indicates the extent that the VM will be affected by co-location.

We propose a profile prediction model that constructs a new stack distance profile for a VM when it shares LLC with other VMs. When VM i and VM j compete for a cache line (which is the d^{th} position in the LRU stack), the probability of VM i "winning" the competition is proportional to the number of accesses to this cache line of VM i, but reversely proportional to the total number of accesses to this cache line of the two VMs. That is,

$$q_d^{ij} = \frac{C_d^i}{C_d^i + C_d^j} \tag{1}$$

We then extend Equ. (1) to the probability of succeeding in accessing the d^{th} position when competing with N_v number of VMs:

$$q_d^i = \frac{C_d^i}{C_d^i + \sum_{k=1}^{N_v} C_d^k}. \tag{2}$$

Then, the new stack distance profile of VM i can be estimated by

$$f_i' = \{q_1^i C_1^i, q_2^i C_2^i, ..., q_A^i C_A^i, C_{>A}'\}, \tag{3}$$

where $C_{>A}' = \sum_{d=1}^{A+1} C_d^i - \sum_{d=1}^{A} q_d^i C_d^i$. The second part in the equation ($\sum_{d=1}^{A} q_d^i C_d^i$) indicates the estimated number of hits of the VM with profile f_i after co-location.

B. Performance Degradation Prediction

Given a group of VMs that are requested by end users, the cloud provider needs to allocate these VMs to available PMs. An effective VM allocation algorithm should allocate as many VMs as possible to a PM while i) meeting their explicit requirements on resources such as CPU and memory and ii) minimizing contentions on LLC. Many previous VM allocation or migration methods provide a metric to choose destination PM and migration VM to handle objective i) but neglect objective ii). In this paper, we focus on designing a metric to measure the performance degradation of the VMs due to their LLC contention to achieve objective ii). Both metrics can be jointly considered to achieve both objectives.

In order to minimize the performance degradation of VMs from LLC contention, we try to avoid co-locating VMs with intensive cache consumptions so that the average impact of cache contention on each VM in the system is minimized. In this section, we first introduce a method to calculate the cache sensitivity and the cache intensity of a VM. By referring to [26], we then propose a pain prediction algorithm that measures the total performance degradation of co-locating two VMs.

Sensitivity is a measure of how much a VM will suffer when cache space is taken away from it due to contention. Intensity is a measure of how much a VM will hurt others by taking away their space in a shared cache. The sensitivity and intensity can be obtained from stack distance profiles. By combining the sensitivity and intensity of two VMs, the pain of co-locating the two VMs can be estimated. Co-locating a sensitive VM with an intensive VM should result in a high level of pain, and co-locating an insensitive VM with any type of VMs should result in a low level of pain for this VM.

When VM i is co-located with VM j, the probability of its hit in the d^{th} position becoming a miss is q_d^{ij} (Equ. (1)). Then, the cache sensitivity of VM i when co-locating with VM j is calculated by

$$S_{ij} = \sum_{d=1}^{A} q_d^{ij} \times C_d^i. \tag{4}$$

Cache intensity of a VM is a measure of how aggressively a VM uses cache. It approximates how much space the VM will take away from its being co-located with VMs. The cache intensity of VM i when being co-located with VM j is calculated by:

$$I_{ij} = S_{ij} + C'_{>A}, \tag{5}$$

which is measured by the number of LLC accesses per one million instructions.

The cache sensitivity and cache intensity are combined into the metric for measuring performance degradation resulting from VM co-location. The VM performance degradation is defined as $\frac{T_{co}-T_{solo}}{T_{solo}}$, where T_{co} and T_{solo} are the total running time of an application in a VM when the VM runs with and without other co-located VMs, respectively. Suppose two VMs v_i and v_j share the same cache. Then, the performance degradation of v_i suffered from being co-located with v_j is calculated by multiplying the intensity of v_j and the sensitivity of v_i when co-located with v_j (i.e., $S_{ij}I_{ji}$). The rationale behind multiplying the intensity and the sensitivity is that combining a sensitive VM with an intensive VM should result in a high level of performance degradation, and combining an insensitive VM with a non-intensive VM should result in a low level of performance degradation. As v_i and v_j are co-located, it is necessary to consider the performance degradation introduced to both v_i and v_j. The degradation of co-scheduling v_i and v_j together is the sum of the performance degradation of the two VMs:

$$P_{ij} = S_{ij}I_{ji} + I_{ij}S_{ji}. \tag{6}$$

V. VM Placement and Migration Algorithm

A. Notations and Assumptions

In this section, we design the CacheVM algorithm to place VMs on PMs, which already hosts a number of VMs, and to migrate VMs from overloaded PMs. We regard it as a VM placement and migration problem.

In practice, the VM placement policy should consider multiple factors simultaneously besides the cache contention, such as load balance and energy saving. Hence, the VM placement policy should be a multi-objective optimization problem. In this work, we focus on minimizing the VM average performance degradation due to cache contention and model the problem. The model can be easily extended to incorporate other objectives. We consider different numbers of VM slots in different PMs, and hide the details about the variance in CPU and memory resource on different PMs as well as the variance in the requested resource from different VMs. The proposed model can be extended to deal with server heterogeneities.

Assume there are N_p PMs in the datacenter, denoted by p_k ($k = 1, 2, ..., N_p$). The k^{th} PM has c_k CPU cores in total, of which t_k are occupied. There are N_{v^a} new VMs to be allocated, namely v_i^a ($i = 1, 2, ..., N_{v^a}$), and they have cache distance profiles f_i, where

TABLE I: Notations and definitions

Notation	Definition
q_d^i	Probability of VM i "winning" the d^{th} position in LRU stack
N_p	The number of PMs
N_{v^a}	The number of VMs to allocate
N_{v^e}	The number of existing VMs
$N_{v^e}^k$	The number of existing VMs in p_k
N_v	Total number of VMs including both existing and new VMs
p_k	PM k
v_i	VM i
v_i^a	New allocated VM i
v_j^e	Existing VM j
c_k	The number of CPU cores in p_k
f_i	Cache distance profile of v_i
f'_i	Predicted cache distance profile of v_i
π	Existing VM to PM mapping
ϕ	VM to PM mapping for new VMs
P_{ij}	The pain due to co-location of v_i and v_j
$\mathcal{P}$	Total pain of allocating new VMs

$$f_i : d \rightarrow C_d^i, \qquad d = 1, 2, ..., A+1.$$

Assume there are N_{v^e} existing VMs allocated in the system, denoted by v_j^e, ($j = 1, 2, ..., N_{v^e}$). The locations of existing VMs are given by the following mapping function:

$$\pi : [v_1^e, v_2^e, ..., v_{N_{v^e}}^e] \rightarrow [p_1, p_2, ..., p_{N_p}]$$

which means that existing VM v_j^e ($j = 1, 2, ..., N_{v^e}$) is located on server $\pi(v_j^e)$.

Then, the pain of the co-location of new VM v_i and existing VM v_j^e can be obtained by Equ. (6). For any valid VM placement trial, there is a corresponding function:

$$\phi : [v_1, v_2, ..., v_{N_{v^a}}] \rightarrow [p_1, p_2, ..., p_{N_p}]$$

which maps the N_{v^a} new VMs to the N_p PMs.

Our optimization goal is to minimize the total pain of the co-location of the new VMs with the existing VMs, denoted by $\mathcal{P}$: Only the VMs that are located in the same PM contribute to the total pain. Then, $\mathcal{P}$ is calculated by:

$$\mathcal{P} = \sum_{i=1}^{N_{v^a}} \sum_{\substack{j=1 \\ \phi(v_i)=\pi(v_j^e)}}^{N_{v^e}} P_{ij} \tag{7}$$

The VM placement problem is then transformed as finding a VM-PM mapping schedule for new VMs (denoted by ϕ) which minimizes the total pain $\mathcal{P}$, under the following constraints:

$$N_{v^a} \leq \sum_{k=1}^{N_p} (c_k - t_k) \tag{8}$$

$$x_{ik} = \begin{cases} 1 & \phi(v_i) = p_k \\ 0 & \phi(v_i) \neq p_k \end{cases}, \forall i = 1, 2, ..., N_{v^a}, k = 1, 2, ..., N_p \tag{9}$$

$$t_k + \sum_{i=1}^{N_{v^a}} x_{ik} \leq c_k, \forall k = 1, 2, ..., N_p \tag{10}$$

$$\sum_{k=1}^{N_p} x_{ik} = 1, \forall k = 1, 2, ..., N_p \tag{11}$$

Constraint (8) guarantees that the number of new VMs does not exceed the total available slots in the datacenter. Constraint (9) is a boolean matrix $[x_{ik}]_{N_{v^a} \times N_p}$ to represent the VM placement solution, where $x_{ik} = 1$ if VM v_i is placed onto PM p_k, (i.e., $\phi(v_i) = p_k$). Constraint (10) guarantees that the number of VMs placed in a PM p_k does not exceed its available slots. Constraint (11) means that each VM can only be placed on one PM.

In the next section, we will transform the optimization problem to an integer linear programming (ILP) model, which can be solved with existing programming toolkits.

B. Integer Linear Programming

In order to solve the problem using an ILP model, we use v_i $(1 = 1, 2, ..., N_v)$ to represent all the VMs, where $N_v = N_{v^a} + N_{v^e}$.We rewrite the optimization goal Equ. (7) as

$$\mathcal{P} = \sum_{i=1}^{N_v} \sum_{j=1}^{N_v} \sum_{k=1}^{N_p} x_{ik} x_{jk} P_{ij} \tag{12}$$

where P_{ij} is the pain of co-locating v_i and v_j as calculated by Equ. (6). Given the stack distance profiles of existing and new VMs, we can calculate the co-location pain of each pair of VMs and finally derive the $N_v \times N_v$ matrix $[P_{ij}]$.

The optimization goal aims to minimize the total pain for every VM co-location. The ILP model requires the object function to be linear, while $x_{ik}x_{jk}$ is nonlinear. We relax and transform the optimization goal by introducing a new variable $y_{ijk} = x_{ik}x_{jk}$. $y_{ijk} = 1$ means that VM v_i and VM v_j are co-located on PM p_k, while $y_{ijk} = 0$ means VM v_i and VM v_j are not co-located on PM p_k. It does not necessarily mean that v_i and v_j are not co-located (e.g., v_i and v_j may be co-located on other PMs other than p_k).

From the definition of y_{ijk}, we can derive the following properties:

1). $y_{ijk} = y_{jik}$;
2). $y_{iik} = 1$ if and only if v_i is on p_k;
3). If $y_{ijk} = 1$, then $y_{iik} = 1$ and $y_{jjk} = 1$;
4). $y_{ijk} \le x_{ik} + x_{jk}$;
5). $y_{ijk} \ge x_{ik} + x_{jk} - 1$.

The problem can be reformulated as below: Given y_{iik}, where $N_{v^a} + 1 \le i \le N_v$, and the profiles f_i $(1 \le i \le N_v)$ of the VMs, find the solution of y_{iik}, where $1 \le i \le N_{v^a}$, that

$$Maximize: \sum_{i=1}^{N_v} \sum_{j=1}^{N_v} \sum_{k=1}^{N_p} y_{ijk} P_{ij}$$

subject to:

$$y_{ijk} = \begin{cases} 1 & \phi(v_i) = \phi(v_j) = p_k \\ 0 & else \end{cases} \tag{13}$$

$$\sum_{i=1}^{N_v} \sum_{j=1}^{N_v} y_{ijk} \le c_k^2 \tag{14}$$

$$\sum_{k=1}^{N_p} y_{ijk} \le 1 \tag{15}$$

$$y_{ijk} = y_{jik}, \quad \forall\ i, j, k \tag{16}$$

$$y_{ijk} \le y_{iik} + y_{jjk}, \quad \forall\ i, j, k \tag{17}$$

$$y_{ijk} \ge y_{iik} + y_{jjk} - 1, \quad \forall\ i, j, k \tag{18}$$

$$\sum_{i=1}^{N_v} \sum_{k=1}^{N_p} y_{iik} = N_v \tag{19}$$

Equ. (14) means that for every PM, there are at most c_k co-located VMs. Equ. (15) means that v_i and v_j can be co-located in at most one PM. Equ. (16), Equ. (17) and Equ. (18) are derived from Properties 1), 4) and 5), respectively. Equ. (19) guarantees that there are in total N_v VMs.

The VM placement problem that considers cache contention between new VMs and existing VMs in the datacenter is NP-hard. A naive way to solve the ILP is to simply remove the constraint that y_{ijk} is integral, solve the corresponding LP (LP relaxation of the ILP), and then round the entries of the solution to the LP relaxation. Here, we can apply the branch and bound method [27] to solve the problem. We use *lpsolve* 5.5 [28] tool to find the optimal solution for the VM allocation.

The computational complexity of the above method is very high, especially for a relatively large number of VMs. Then, we propose a heuristic VM placement and migration algorithm below.

Algorithm 1 Greedy VM placement algorithm

```
Input: $N_{v^a}$ VMs, each with distance profile $f_i$ $(1 \le i \le N_{v^a})$
       $N_{v^e}$ existing VMs, each with profile $f_j$ $(1 \le j \le N_{v^e})$
       Locations of existing VMs $\pi$
Output: a feasible placement schedule $\phi$ for the VMs
 1: Compute the cache hits based on $f_i$ $(1 \le i \le N_{v^a})$ for each VM;
 2: Sort the VMs based on cache hits in decreasing order;
 3: Compute the total cache hits based on existing VMs $v_j^e$ for each PM;
 4: Sort the PMs based on the total cache hits in ascending order;
 5: for each VM $v_i^a$ in the sorted list do
 6:   for each PM $p_k$ do
 7:     Estimate the stack distance profile of $v_i^a$ in $p_k$ (Equ. (3));
 8:     Compute $P_{v_i^a}^k$ based on Equ. (20);
 9:     if $P_{v_i^a}^k < p_{min}$ then
10:       $p_{min} \leftarrow P_{v_i^a}^k$;
11:       $p_{dest} \leftarrow p_k$;
12:     end if
13:   end for
14:   place $v_i^a$ on $p_k$;
15: end for
16: Return $\phi$;
```

VM placement. Given N_{v^a} VMs with their stack distance profiles f_i $(1 \le i \le N_{v^a})$, our greedy VM placement heuristic allocates each VM to a PM that leads to the minimum total performance degradation, i.e., pain ($P_{v_i^a}^k$).

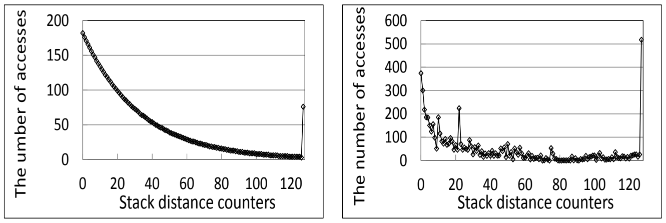
$$P_{v_i^a}^k = \sum_{\forall\ v_j^e\ in\ p_k} P_{ij}. \tag{20}$$

Algorithm 1 shows the pseudo-code of the greedy VM placement. The algorithm first sorts the VMs based on cache hits derived from the input profiles in decreasing order (Line 1-2), and then sorts the PMs based on the total cache hits in ascending order (Line 3-4). The total cache hits of a PM p_k is calculated by summing the number of hits of existing VMs in the PM. The algorithm iterates through the PM list (Line 6) to find the PM that will result in the least pain with the VM based on Equ. (20) (Line 7-12). Finally, the algorithm returns a feasible placement ϕ for the VMs.

VM migration. When a PM becomes overloaded, it needs to migrate out VMs to move out its extra load. The VM migration algorithm needs to select migration VMs from an overloaded

Algorithm 2 VM selection algorithm

```
Input: VM list with profiles in PM
Output: VM for migration
1: for VM $v_i$ in VM list do
2:     $totalPain \leftarrow P^k_{v^e_i}$;
3:     if $totalPain > p_{max}$ then
4:         $p_{max} \leftarrow P^k_{v^e_i}$
5:         $vm \leftarrow v_i$;
6:     end if
7: end for
8: Return $vm$;
```



(a) Random profile (b) Profile of NPB benchmark LU

Fig. 3: Stack distance profiles.

PM and select the destination PM to host each migration VM. The basic idea of the VM section algorithm is to select a VM which generates the maximum pain with other co-located VMs in the PM. We define the pain induced by v^e_i with other co-located VMs in p_k as

$$P^k_{v^e_i} = \sum_{\forall\ v^e_j\ in\ p_k, j \neq\ i} P_{ij}. \tag{21}$$

Algorithm 2 show the migration VM selection procedure. The algorithm iterates through the VMs list in the PM (Line 1). In each iteration, it calculates the pain of this VM with other VMs (Line 2). The VM that results in maximum $totalPain$ is selected (Line 3-6) and returned (Line 8). To find a destination PM to host a migration VM, Algorithm 1 is used to select the PM that leads to the minimum total pain for the migration VM based on Equ. (20).

VI. Performance Evaluation

A. Experiment Settings

We conducted experiments on CloudSim [29], a modern simulation framework for cloud computing environments and on our school's primary high-performance computing (HPC) cluster (a 21,546-core 500 tera FLOPS HPC system). We extended CloudSim to model LLC contention between VMs by adding a stack distance profile to each VM. The profiles are determined by the trace. The simulator then takes the profiles as input to predict VM miss rate due to co-location based on our proposed cache contention model. We simulated a datacenter that consists of more than 1000 physical nodes. Each node is emulated to be equipped with HP ProLiant DL380 G5 (1 x [Xeon 3075 3160 MHz, 4 cores], 4GB). The characteristics of the VM types correspond to Amazon EC2 instance types [30]: (2.5GHz, 0.85GB), (2.0GHz, 3.75GB), (1.0GHz, 1.7GB) and (0.5GHz, 613 MB).

To study the performance interference caused by contention for LLC quantitatively, we conducted the simulation experiments based on both random stack distance profiles (created by ourselves) and trace-driven profiles. We designed the random stack distance profiles rather than only using the trace-driven profiles in order to evaluate the performance for different types of cache access behaviors.

Random stack distance profiles. Intuitively, a VM's number of hits to its Most Recently Used (MRU) positions is larger than its number of hits to the LRU positions in a cache. In the random profile based experiment, we created an exponential model to create a VM's stack distance profile that emulates its stack access behavior as shown in Figure 2(a):

$$C_d = a \times e^{\frac{-(\frac{1}{b}) \times d}{0.005 \times c}}$$

where a, b and c are coefficients used to control the shape of the profile. a controls the number of hits to the MRU position (i.e., the height of the curve in Figure 2(a)) and b and c together control the decreasing speed of the curve (i.e., the number of hits from the left (MRU) to the right (LRU) in Figure 2(a)). A high value of b or c means a slower decreasing speed. Figure 3(a) shows an example of a random profile with $a = 180$, $b = 40$ and $c = 200$. In the simulation, we randomly selected the values for a, b and c in a certain range to generate the profiles.

Experiments on trace-driven profiles. In the trace-driven experiment, we selected workloads from the NAS Parallel Benchmark (NPB) suite [31]. The NPB suite is a small set of programs (as shown in Table II) designed to help evaluate the performance of parallel supercomputers. It provides a variety of memory accessing applications. The workload size of each program in NPB can be specified as different classes (e.g., small, standard, large, etc.) before running. We used the MPI implementation of the NPB, Version 3.3 (NPB3.3-MPI). We executed the programs on our HPC cluster and used PinTools [32] to record the stack distance profiles for the programs. We then used the measured stack distance profiles in our trace-driven experiments. We randomly chose a profile from the 8 profiles for each VM in our experiments.

Figure 3(b) shows an example of the measured profile of NPB benchmark LU with a small workload. Unlike the random profile in Figure 3(a), the real profile can have a higher number of accesses for a counter with a high distance than those with small distance (e.g., the number of accesses for C_{22} is greater than the number for C_{20}). Furthermore, the number of cache misses in the trace is greater than the random profile.

Model validation. In order to study the accuracy of the proposed model in predicting the behavior of real applications, we conducted an evaluation experiment. This experiment is performed using a execution-driven simulator [33]. We chose paired benchmarks from Table II and co-scheduled them to the simulator. 28 benchmark pairs were studied in the experiment. We measured the number of cache misses of the benchmarks when they ran under cache sharing. We also predicted the number of cache misses

TABLE II: NAS Parallel Benchmarks

Application	Specifications
IS	Integer Sort, random memory access
EP	Embarrassingly Parallel
CG	Conjugate Gradient, irregular memory access and communication
MG	Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
FT	discrete 3D fast Fourier Transform, all-to-all communication
BT	Block Tri-diagonal solver
SP	Scalar Penta-diagonal solver
LU	Lower-Upper Gauss-Seidel solver

based on our model. We then computed the cache miss prediction errors, which are the difference in the cache misses predicted by the model and collected by the simulator under cache sharing, divided by the number of cache misses collected by the simulator under cache sharing. Figure 4 shows the cumulative distribution function of the prediction errors. We see that 80% of the predictions have prediction error lower than 6%. The model achieves a median prediction error of 2%. This result confirms that the proposed model achieves a high accuracy in predicting cache behaviors.

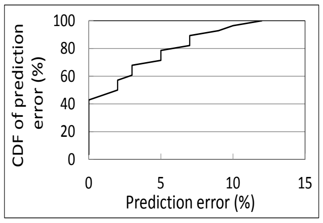
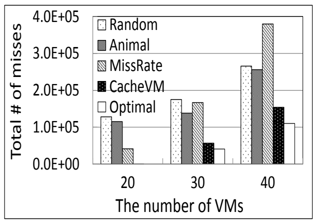


Fig. 4: Prediction error.

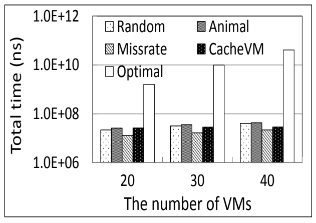
Comparison algorithms. In order to compare the performance of *CacheVM* with other alternatives, we also implemented new VM placement algorithms in CloudSim based on CacheVM (denoted by *CacheVM*), cache contention unaware algorithm (denoted by *Random*), classification based algorithm [17] (denoted by *Animal*) and miss rate based algorithm [19] (denoted by *MissRate*), respectively. *Random* algorithm randomly selects a PM and assigns a VM to the PM as long as the PM has enough capacity (e.g., CPU, memory, bandwidth) to host the VM. In VM migration, it randomly selects VMs and migrates them to randomly selected PMs. *Animal* algorithm classifies the VMs to four classes based on the algorithm introduced in [17] and avoids co-locating two VMs with type *Devil* on purpose in both VM placement and migration. *MissRate* [19] algorithm estimates the miss rate of each VM and then selects the PM that results in the least total miss rate of its existing VMs to be the destination of a given VM. For the performance evaluation of VM placement, we allocated VMs to PMs according to each allocation algorithm, and then measured the total number of misses in the system after the program executions completed without considering VM migration.

B. Trace-driven Comparison with the Optimal Algorithm

Since the computational complexity of the integer linear programming algorithm is high, it is difficult to solve the problem with a relatively larger number of VMs. In order to examine how the proposed ILP optimization algorithm performs in mitigating cache contention, we carried out the VM placement experiment in a small scale with 20 PMs and a varied number of VMs from 20 to 40. Figure 5(a) shows the total number of misses of all the VMs in the system. The result with the ILP optimization algorithm is denoted as *Optimal*. We see that the result follows *Optimal<CacheVM<Animal<Random*. It indicates that *Optimal* outperforms all the other algorithms for solving small scale problems. *CacheVM*'s performance is very close to *Optimal* and higher than other algorithms. The number of misses of *MissRate* increases faster under different numbers of VMs than the other algorithms. This is because *MissRate* does not consider the contention between co-located VMs by aiming to minimizing the total miss rate of VMs in a PM. *Random* generates the highest number of misses because it does not consider cache contention in VM placement. *Animal* improves *Random* since it avoids co-locating VMs with serious cache contention. However, due to its cause classification of VMs, it has higher number of misses than *CacheVM*.



(a) Total number of misses



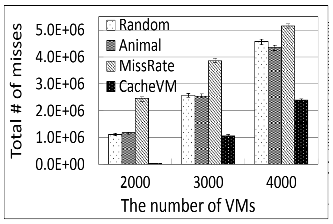
(b) Computation time

Fig. 5: Trace-driven comparison with the optimal solution.

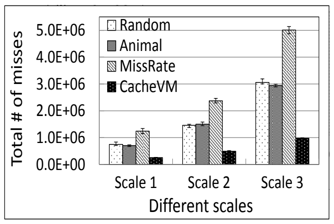
In order to study the runtime overhead of the proposed algorithms, we tested the computation time. The experiment was conducted on a server with 3.00GHz Intel(R) Core(TM)2 CPU and 4GB memory. Figure 5(b) presents the CPU time in nanoseconds in logarithm scale for each algorithm. We see that the computation time follows *MissRate<CacheVM<Random<Animal<<Optimal*. Specifically, *MissRate*, *CacheVM*, *Random*, *Animal* and *Optimal* consume 0.041, 0.042, 0.022, 0.029 and 40.908 seconds, respectively. *MissRate* consumes the least time because it finds PMs to host VMs simply based on the total miss rate of VMs in each PM without extensive computation. *Random* consumes more time than *CacheVM* because it must launch extra runs when the random VM to PM mapping is infeasible (i.e., the PM does not have enough capacity). *Animal* consumes more time than the previous three algorithms because it involves a relatively complex procedure to classify the VMs. *Optimal* consumes much more time than the other algorithms due to the high complexity of solving the ILP problem. As the computational complexity of *Optimal* is too high, *Optimal* is not suitable for large-scale systems. In the following, we present the performance of *CacheVM* in comparison with other algorithms in large-scale systems.

C. Performance with Random Profiles

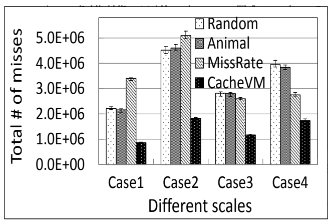
In this experiment, the stack distance profiles for the VMs are generated with a randomly selected in $[0, 200]$, b in $[50, 100]$, c in $[0, 200]$ unless otherwise specified. We set the number of PMs to 2000, and varied the number of VMs from



(a) Different number of VMs on 2000 PMs

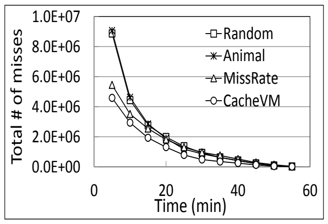


(b) Different scales

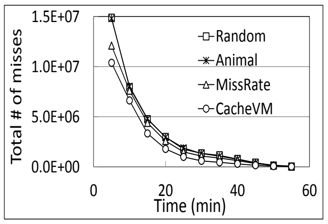


(c) Different profile models

Fig. 6: Performance in VM placement with random profiles.



(a) 2000 PMs and 3000 VMs



(b) 3000 PMs and 4000 VMs

Fig. 7: Performance in VM migration with random profiles.

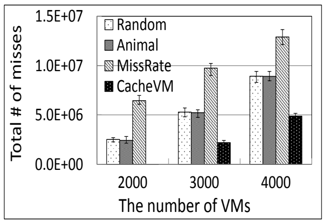
2000 to 4000. Figure 6(a) shows the median, the 10^{th} and 90^{th} percentiles of the total number of misses after the initial placement of the VMs. We see that the total number of misses in the system follows *CacheVM*<*Animal*<Random<*MissRate*. The experimental result also shows that the variance follows this order although it is not very obvious in the figure. *MissRate* performs the worst due to its insufficiently accurate cache contention estimation algorithm merely based on the cache miss rate. VM placement algorithm based on cache miss rate is not able to efficiently relieve the cache contention problem. *Random* does not perform as well as the other two cache contention aware algorithms because it randomly allocates VMs to PMs without considering cache interference. *Animal* performs worse than *CacheVM* due to the reason that the coarse classification of VMs cannot accurately reflect actual cache interference between VMs hence cannot effectively avoid interference between VMs. We can also see that the total number of misses increases as the VM to PM ratio increases (i.e., as more VMs are allocated in the system), there is a higher chance for cache intensive VMs being co-located. When the VM to PM ratio is 1, *CacheVM* is able to achieve the minimum number of misses, while the others still lead to an extra number of misses besides the minimum number of misses in the system.

We then measure the total number of misses using different datacenter scales with the same VM/PM ratio including (1000 VMs, 750 PMs), (2000 VMs, 1500 PMs) to (4000 VMs, 3000 PMs). Figure 6(b) shows the median, the 10^{th} and 90^{th} percentiles of the total number of misses with different datacenter scales. We see that the total number of misses in the system follows *CacheVM*<*Animal*<Random<*MissRate*, due to same reasons explained before.

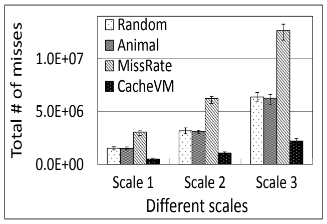
Finally, we study the performance with different stack distance profiles. We set the number of PMs and VMs to 3000 and 4000, respectively. We designed four categories of profiles with different a and b values (randomly selected from a range) in the profile generator, representing different types of cache visiting behaviors. The settings for the four categories are:
Case 1: $a \in [0, 50]$, $b \in [20, 50]$; low MRU hits, slow drop;
Case 2: $a \in [0, 100]$, $b \in [20, 50]$; high MRU hits, slow drop;
Case 3: $a \in [0, 50]$, $b \in [50, 80]$; low MRU hits, median drop;
Case 4: $a \in [0, 50]$, $b \in [80, 100]$; low MRU hits, fast drop.
We set $c = 200$ for all cases.

Figure 6(c) presents the median, the 10^{th} and 90^{th} percentiles of the results from different types of profiles. We see that the total number of misses in the system follows *CacheVM*<*Animal*<*Random*<*MissRate* for Case 1 and Case 2 due to the same reasons mentioned before. The total number of misses in the system follows *CacheVM*<*MissRate*<*Animal*<*Random* for Case 3 and Case 4. *MissRate* has different performance for different profile settings, while others maintain relatively stable performance. This is because *MissRate* finds VM-PM mapping solutions based on the miss rate metric, which is sensitive to the types of profiles. In all the four cases, *CacheVM* produces the fewest misses.

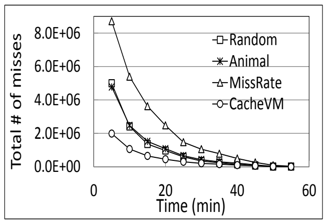
In order to study the performance of the VM migration algorithms, we extend the previous experiment by deliberately conducting VM migrations in the system every 5 minutes after VM placement. Each VM's workload was randomly generated. A PM executes the migration VM selection algorithm to select VM(s) to migrate out, and then the centralized server executes the destination PM selection algorithm to find destinations for the migration VMs. We also measured the total number of misses in the system every 5 minutes. In this experiment, we test two scenarios with settings of (3000 VMs, 2000 PMs) and (4000 VMs, 3000 PMs), respectively. The settings are the same as Figure 6(a) except that we used a in $[0, 200]$, b in $[50, 100]$ to generate the profiles. Figure 7 shows the total number of misses as a function of time for the two testing cases correspondingly. In both figures, we see that after initial VM placement, the total number of misses of all the algorithms decreases due to the reason that the algorithms repeatedly migrate VMs out from PMs with high cache contention to PMs with low cache contention. After several rounds of VM migrations (e.g., after 40 minutes), the total number of misses of each algorithms still follows *CacheVM*<*MissRate*<*Animal*≈*Random*. The reasons of this performance order is explained in Case 4 in Figure 6(c). After several rounds of VM migration, *CacheVM* still has fewer misses than the other algorithms although it is not very obvious in the figure.



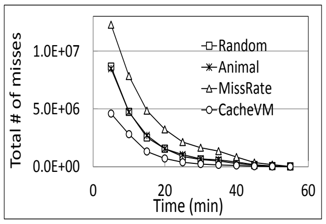
(a) Different number of VMs on 2000 PMs



(b) Different scales



(c) 2000 PMs and 3000 VMs



(d) 3000 PMs and 4000 VMs

Fig. 8: Performance in VM placement and migration in trace-driven simulation.

D. Performance with Real Trace

In this section, we study the performance of the algorithms with real trace. In these experiments, the environment settings are the same as the previous section.

Figure 8(a) shows the median, the 10^{th} and 90^{th} percentiles of the total number of misses under varying VM to PM ratio. We see that the total number of misses in the system follows *CacheVM<Animal≈Random<MissRate*. Compared to the result in Figure 6(a) for the random profiles, *Animal*'s performance is still higher but closer to *Random* because merely avoiding co-locating two *Devil* VMs cannot guarantee minimizing the total number of misses in the system. *CacheVM* still achieves a much lower total number of misses than the other algorithms. The reasons for these results are the same as Figure 6(a).

Figure 8(b) shows the median, the 10^{th} and 90^{th} percentiles of the total number of misses in the system in different scales presented in Section VI-C. We see that the total number of misses in the system follows *CacheVM<Animal<Random<MissRate*, which is consistent with previous results in Figure 8(a) due to the same reasons.

Figure 8(c) and Figure 8(d) show the total number of misses as a function of time. The procedure of the experiment is the same as in Figure 7. We see that after initial VM placement, the total number of misses of all algorithms decreases due to the reasons mentioned in Figure 7. At all rounds of VM migrations, the total number of misses of each approach follows *CacheVM<Animal≈Random<MissRate* because of the same reasons in Figure 8(a). The result confirms that *CacheVM* outperforms the other algorithms in the real trace.

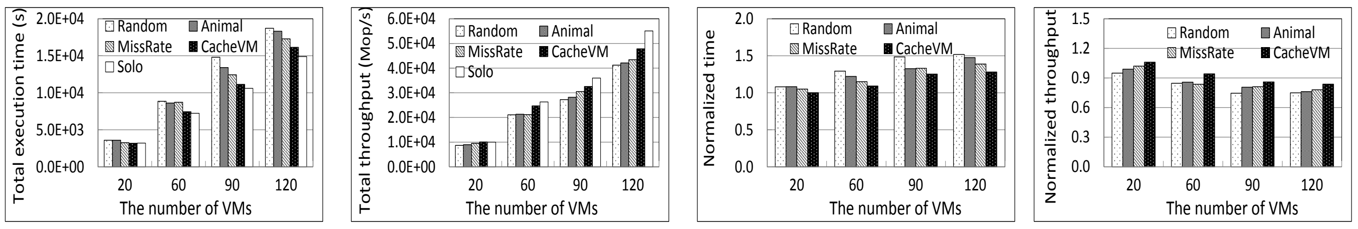
E. Trace-driven Experiments on Real Testbed

We also carried out the experiment on our school's HPC cluster. We deployed the experiment with 20 physical nodes. Each node is equipped with Intel Xeon E5345, 12GB main memory, and contains 2 quad-core processors. The cores in each processor share a 4MB, 16-way set associative LLC with a 64-byte cache line. We used the corresponding algorithms (i.e., *CacheVM*, *Animal*, *MissRate*) to conduct VM allocation experiments based on our obtained real trace profiles. Each VM ran a randomly selected program from the NPB suite with the medium workload size (i.e., class A). We measured the execution time and throughput after all program executions were completed. Besides comparing the performance of different VM placement algorithms, we also measured the performance of each VM when it exclusively ran on a physical node. In this case, the VM could use all the cache resources, and hence its performance was not degraded by cache contention. We denote the results from the exclusive running as Solo and present these results as the optimal performance. We varied the number of VMs from 20 to 120 and allocated them to 20 PMs.

Figure 9(a) shows the total execution time of all the VMs. We see that all the algorithms perform relatively well when the number of deployed VMs is small (20 VMs) and they produce a similar total execution time as the optimal total time. This is because there are sufficient resources for the VMs and cache contention rarely occurs. When the number of VMs increases, the total execution time of different algorithms follows *CacheVM<MissRate<Animal<Random*. *CacheVM* outperforms the other algorithms because it more accurately estimates the performance degradation of the VMs due to co-location, appropriately separates the VMs severely interferences each other to different PMs, and hence minimizes the total performance degradation. The result is consistent with the result of Case 4 in Figure 6(c) because of the same reasons.

Figure 9(b) shows the total throughput of all the VMs. The total throughput of Sole is calculated by checking the programs running in the VMs and assuming every one of them delivers optimal throughput. Compared to Solo, all the algorithms experience performance degradation due to cache contention between VMs. Among the four VM placement algorithms, the throughput follows *Random<Animal<MissRate<CacheVM* due to the same reasons demonstrated before.

In order to investigate how the placement algorithms influence the performance of individual VMs, we first normalized the execution time and the throughput of each VM to the optimal execution time and throughput of the VM, and then calculated the average normalized execution time and throughput per VM. Figure 9(c) shows the average normalized execution time of the VMs. Similarly, the result follows *CacheVM<Animal<MissRate<Random*. *CacheVM* improves the cache contention-unaware algorithm (i.e., *Random*) with 9.0%, 14.2%, 26.6% and 17.0% lower normalized total execution time for the experiment with 20, 60, 90 and 120 VMs, respectively. Figure 9(d) shows the average normalized throughput per VM. The average normalized throughput follows *Random<Animal<MissRate<CacheVM*. *CacheVM* improves *Random* with 6.0%, 5.1%, 7.3% and 4.7% higher average normalized throughput for the four test cases. The orders of the performances of Figure 9(c) and Figure 9(d) are consistent with Case 3 and Case 4 in Figure 6(c) due to the same reasons.



(a) Total execution time (b) Total throughput (c) Avg. normalized execution time per VM (d) Avg. normalized throughput per VM

Fig. 9: Performance of real-testbed experiments.

VII. Conclusion

Cache contention in LLC can lead to serious performance degradation on VMs in a cloud datacenter. Previous VM placement and VM migration algorithms neglect such cache interference. In this paper, we proposed a cache contention aware VM performance degradation prediction algorithm based on stack distance profiles. Based on the estimation, we formulated a cache contention aware VM placement problem to minimize the total performance degradation. We transformed this problem to an integer linear programming (ILP) model and solved it with existing programming toolkits. We then proposed a heuristic cache contention aware VM placement and migration algorithm, called *CacheVM*, as a problem solution. Trace-driven simulation and real-testbed experiments show that *CacheVM* reduces average performance degradation experienced by the VMs due to cache contention and improves the execution time and throughput compared with previous methods. In the future, we will develop a decentralized version of the proposed algorithm to make it scalable to large-scale cloud datacenters.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, IBM Faculty Award 5501145 and Microsoft Research Faculty Fellowship 8300751.

References

[1] "Amazon web service," http://aws.amazon.com/.
[2] "Google Cloud," https://cloud.google.com/.
[3] "Microsoft Cloud," http://www.microsoft.com/enterprise/microsoft cloud/default.aspx#fbid=MlUrRhT5amn.
[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. of NSDI*, 2005, pp. 273–286.
[5] "Service Level Agreements," http://azure.microsoft.com/en-us/support/legal/sla/.
[6] L. Chen, H. Shen, and K. Sapra, "RIAL: Resource intensity aware load balancing in clouds." in *Proc. of INFOCOM*, 2014.
[7] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing." in *Proc. of HotPower*, 2008.
[8] Y. Hong, J. Xue, and M. Thottethodi, "Dynamic server provisioning to minimize cost in an IaaS cloud." in *Proc. of SIGMETRICS*, 2011.
[9] L. Chen and H. Shen, "Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters." in *Proc. of INFOCOM*, 2014.
[10] Y. Lin, H. Shen, and L. Chen, "Ecoflow: An economical and deadline-driven inter-datacenter video flow scheduling system," in *Proc. of Multimedia*, 2015, pp. 1059–1062.
[11] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems." in *Proc. of SOCC*, 2011.
[12] L. Chen, H. Shen, and K. Sapra, "Distributed autonomous virtual resource management in datacenters using finite-markov decision process," in *Proc. of SOCC*, 2014, pp. 1–13.
[13] L. Yu, L. Chen, Z. Cai, H. Shen, Y. Liang, and Y. Pan, "Stochastic load balancing for virtual resource management in datacenters," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2016.
[14] L. Chen, S. Patel, H. Shen, and Z. Zhou, "Profiling and understanding virtualization overhead in cloud," in *Proc. of ICPP*, 2015, pp. 31–40.
[15] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer," in *Proc. of ISM*, 2008.
[16] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *Proc. of IISWC*, 2006, pp. 225–236.
[17] Y. Xie and G. Loh, "Dynamic classification of program memory behaviors in CMPs," in *Proc. of CMP-MSI*, 2008.
[18] H. Jin, H. Qin, S. Wu, and X. Guo, "CCAP: a cache contention-aware virtual machine placement approach for HPC cloud," *IJPP*, pp. 1–18, 2013.
[19] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS observations to improve performance in multicore systems," *IEEE micro*, no. 3, pp. 54–66, 2008.
[20] S. Kim, H. Eom, and H. Y. Yeom, "Virtual machine consolidation based on interference modeling," *SC*, 2013.
[21] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proc. of SigComm*, 2015, pp. 407–420.
[22] A. Rai, R. Bhagwan, and S. Guha, "Generalized resource allocation for the cloud," in *Proc. of SOCC*, 2012.
[23] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proc. of HotCloud*, 2012.
[24] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu, "A-drm: Architecture-aware distributed resource management of virtualized clusters," in *Proc. of VEE*, 2015, pp. 93–106.
[25] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *Proc. of SC*, 2001.
[26] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of SIGARCH CAN*, vol. 38, no. 1, 2010, pp. 129–142.
[27] V. I. Norkin, G. C. Pflug, and A. Ruszczyński, "A branch and bound method for stochastic global optimization," *Mathematical programming*, vol. 83, no. 1-3, pp. 425–450, 1998.
[28] "lpsolve 5.5," http://lpsolve.sourceforge.net/5.5/Java/README.html.
[29] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." *SPE*, 2011.
[30] "EC2 Instance types," http://aws.amazon.com/ec2/instance-types/.
[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Proc. of SC*, 1991, pp. 158–165.
[32] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of ACM Sigplan*, 2005.
[33] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.