Interference Management for Distributed Parallel Applications in Consolidated Clusters

Jaeung Han*, Seungheun Jeon*, Young-ri Choi†, and Jaehyuk Huh*

School of Computing, KAIST*, School of Electrical and Computer Engineering, UNIST†

{juhan,shjeon,jhuh}@calab.kaist.ac.kr, ychoi@unist.ac.kr

Abstract

Consolidating multiple applications on a system can improve the overall resource utilization of data center systems. However, such consolidation can adversely affect the performance of some applications due to interference caused by resource contention. Despite many prior studies on the interference effects in single-node systems, the interference behaviors of distributed parallel applications have not been investigated thoroughly. With distributed applications, a local interference in a node can affect the whole execution of an application spanning many nodes. This paper studies an interference modeling methodology for distributed applications to predict their performance under interference effects in consolidated clusters. This study first characterizes the effects of interference for various distributed applications over different interference settings, and analyzes how diverse interference intensities on multiple nodes affect the overall performance. Based on the characterization, this study proposes a static profiling-based model for interference propagation and heterogeneity behaviors. In addition, this paper presents use case studies of the modeling method, two interference-aware placement techniques for consolidated virtual clusters, which attempt to maximize the overall throughput or to guarantee the quality-of-service.

Categories and Subject Descriptors C.1.4 [*Parallel Architectures*]: Distributed architectures; C.4 [*Performance of Systems*]: Modeling techniques; D.4.1 [*Process Management*]: Scheduling

Keywords Resource contention, consolidated system, cloud computing, distributed parallel application, interference model, placement algorithm

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA.

http://dx.doi.org/10.1145/2872362.2872388

1. Introduction

An increasing number of cores in multi-core processors have been exacerbating contention on shared architectural resources such as shared last-level caches (LLC) and memory bandwidth. Such contention causes performance interference among applications in consolidated systems where different types of applications share a physical machine. With the growing popularity of cloud computing and system virtualization, such a consolidation of divergent applications on the same system has become common. To mitigate the performance interference problem, recent studies have been exploring schemes to reduce performance variation or to support a certain level of performance guarantee, regardless of the behaviors of co-running applications [7, 13, 15, 16, 21].

However, most of the prior work have isolated the effect of interference within a physical system. Their models assume that the interference occurring in a system affects only the applications running on the same system. For example, in the recent studies to control the interference [13, 21], the performance of a critical application running on a single physical system is protected from co-running batch applications. By estimating the resource contention and performance impact by the co-running applications, the studies have prevented batch applications from reducing the performance of the critical application lower than a threshold.

However, in clouds, another important class of applications are distributed parallel applications, which span multiple physical systems (nodes). In such distributed parallel applications, the effect of interference in a physical node affects the final latency of the applications differently, depending on how their parallelism is utilized. For example, in one type of distributed applications, the interference in one of the nodes can propagate to the entire participating systems, if load balancing is not dynamically adjusted. In this case, one slow node can delay the whole execution of the application, since the other nodes cannot proceed to the next stage until the delayed one is completed. A different type of distributed applications can be resilient to such an isolated interference, minimizing the propagating effect. Furthermore, when multiple nodes suffer from different intensities of interference, the final latency of distributed applications may depend on

how the different levels of interference manifest themselves in the parallel execution.

For distributed applications in consolidated systems, this paper investigates how interference in a subset of nodes affects the final execution times of the applications, exploring techniques to model the interference effect. Unlike prior studies which have been isolating interference estimation within a physical system boundary [13, 21], to the best of our knowledge, this paper is one of the first studies to investigate the interference effect on distributed systems and to propose a method to estimate the effect. Such interference management is important to guarantee the quality of service of distributed applications, or to improve the overall throughput of consolidated clusters.

The proposed interference-aware performance model covers two different aspects of interference manifestation in distributed systems. First, the *interference propagation* model estimates how interference in a subset of nodes determines the final latency. Second, the *interference heterogeneity* model converts different interference intensities in a subset of nodes to a homogeneous intensity in the same or different numbers of nodes, to simplify the estimation model. Combining the two aspects, the proposed technique can estimate the final performance of a distributed application under different interferences in a subset of nodes with a small number of profiling runs.

Based on the proposed modeling technique, this paper investigates interference-aware placement algorithms for distributed applications as a case study. Using the performance model, two algorithms based on a simulated annealing technique find the best placement of multiple distributed applications on a cluster of physical nodes with two different goals. The first algorithm finds the best placement for maximizing the overall throughput. The second algorithm finds the best placement to support a certain level of performance guarantee for an application while improving the throughput of the other applications. Our experimental results show that the placement algorithms can effectively improve the overall performance with quality-of-service (QoS) support using the proposed interference model.

The main contributions of this paper are as follows. First, this paper investigates how the final performance of a distributed application is determined when a subset of the nodes suffer from various intensities of interference. Second, this paper proposes a method to build an interference-aware performance model for distributed applications. Although it requires profiling runs to characterize the interference propagation behaviors of applications, the method reduces the required profiling runs significantly, compared to a naive design. Third, using the proposed model, this paper explores interference-aware placement algorithms for distributed applications on a cluster of physical nodes. By reflecting the interference propagation behaviors, the placement algorithms improve the overall throughput or support QoS for distributed applications.

Limitations: As the first effort to model the interference in distributed applications, this paper assumes that each distributed application is known a priori, and the interference model of the application is constructed with a small number of separate profiling runs. The model from the separate profiling runs cannot be adjusted dynamically during application execution. In addition, it allows to predict the effects of pairwise interaction in each node. In each node, the model assumes that only up-to two distinct applications share the node resources. A similar approach has been used in the prior work [13]. Mars et al. first used a profile-based model construction for pairwise co-location as the initial mechanism of interference management [13]. Their subsequent work extended the profile-based approach to on-line estimation which supports co-locations of multiple applications [21]. Supporting a dynamic model for more than two distributed applications in each node will be our future work.

The rest of the paper is organized as follows. Section 2 describes a prior technique to quantify and model interference in a single node system. Section 3 investigates two aspects, propagation and heterogeneity of interference in distributed systems. Section 4 proposes a modeling method to minimize required profiling runs, and validates our model. Section 5 proposes two interference-aware placement algorithms, and shows the experimental results. Section 6 presents the validation results with a commercial cloud service. Section 7 presents the related work and Section 8 concludes the paper.

2. Background and Motivation

2.1 Prior Techniques for Quantifying Interference

This section presents a prior technique proposed by Mars et al. to model the effect of interference for pairwise applications running on single-node systems [13]. Among various sources of performance interferences, shared last-level caches (LLC) and memory bandwidth have been dominantly contended resources for compute-intensive applications [13, 16, 21]. If one application evicts cachelines for a co-running application or consumes scarce memory bandwidth excessively, the performance of the co-running application is adversely affected. In this paper, we use the effect of shared cache and memory bandwidth interference as the primary source of performance interference, although the technique can be generalized to different types of interferences such as network and disk I/O bandwidth, as discussed by the prior studies [13, 16, 21].

To model the interference behavior of distributed applications, we employ a technique proposed by a prior study, to model the interference behavior within a physical node boundary. The single-node technique called *Bubble-Up*, measures how the performance of each application is degraded by different intensities of interference. Based on the

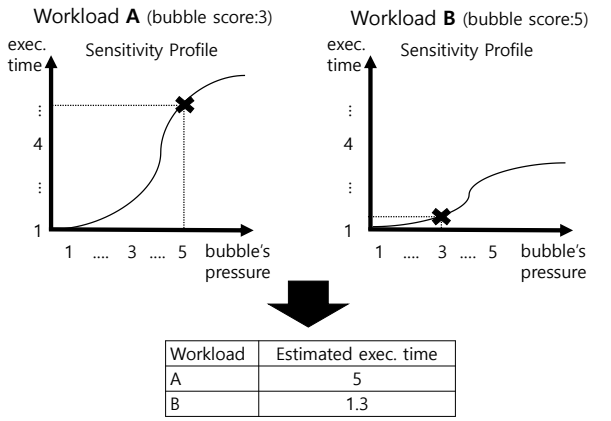


Figure 1. Estimating execution times of consolidated workloads with interference sensitivity curves and bubble scores

performance changes, the technique generates interference response curves (*sensitivity profile*) for each application with profiling runs [13]. In addition to the per-application interference sensitivity profile, it also measures how much interference the application generates to co-runners, and normalizes the generated interference to a score (*bubble score*). Using the per-application sensitivity profile and the bubble score generated by the application, the interference management system can estimate the performance of any two applications when the two applications are scheduled to the same system. The study divides applications to mission-critical tasks and batch ones, and the batch applications are scheduled only when they do not degrade the performance of mission-critical ones beyond a threshold [13, 21].

A key component of the interference modeling technique is to normalize the interference intensity to a bubble score. To generate interference intensity in a controlled manner, the study designs *bubble*, a small interference-generation program, which exercises the memory subsystem by creating different levels of cache misses and external memory accesses. Using the bubble program, the performance of each application is measured with different pre-defined levels of interference. With profiling runs for each application under different interference levels, the scheme measures how much performance degradation the application exhibits under a given level of interference. In Figure 1, the interference sensitivity profile for each workload models the execution time increase pattern by the pressure increase using the co-running *bubble*.

The same bubble program is also used to measure how much interference an application generates. Based on the performance degradation of the bubble program itself, when it is co-run with a target application, the interference intensity the application causes is converted to a bubble score. In essence, the bubble-based approach provides an interference normalization method, so that the interference effect of every pair of co-running applications does not need to be profiled

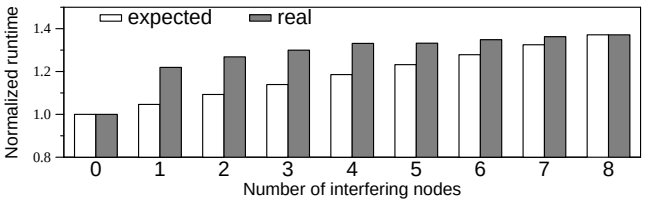


Figure 2. Execution time of 126.lammps over various numbers of nodes executing 462.libquantum

a priori. For each application, the per-application sensitivity profile and bubble score need to be measured, without knowing what another application will be co-run. When two applications are deployed to a system, their possible performance interference can be estimated by examining the sensitivity profiles and bubble scores of the two applications. Figure 1 shows the procedure to estimate the performance of two co-running applications by combining the sensitivity curves and bubble scores.

The subsequent study by Yang et al. extended the bubble-based interference model to on-line measurement which also supports co-locations of multiple applications in a single node [21]. The interference response and generation models of applications are constructed on the fly by executing profiling runs with bubble occasionally. They optimized the occurrence of profiling runs, by tracking changes of the system behavior.

A different prior approach, *DeepDive* uses an interference model with resource usage statistics for interference management [16]. The interference model estimates possible interference and performance degradation, from the resource usage statistics of co-running applications. When a task is submitted, it runs on an isolated virtual machine (VM) without interference to characterize the behavior of the application for a short period of time, collecting key resource usage statistics such as cache misses or I/O accesses. Once the characterization of the application is completed with the short dynamic profiling run, the task is migrated to an appropriate node, by estimating the possible interference by the new application and the existing applications based on the resource usage statistics. Instead of normalizing interference levels to bubble scores, the approach directly models interference with memory and I/O usage statistics. It also exploits the migration capability of virtualization to avoid separate profiling runs, although the effectiveness of the approach relies on the stability of application behaviors since only a short initial run must be able to represent the whole execution.

2.2 Motivation

In this paper, we use the same approach as the bubble-based interference measurement for each individual node participating the execution of distributed applications. Using an interference generation program similar to *bubble*, we generate several different intensities of interference in each node.

The main contribution of this work is how each individual interference in distributed nodes should be combined to determine the final performance of distributed applications.

As a motivating example, Figure 2 presents how the final runtime of a parallel application is determined by interference in a subset or all of the nodes. For the experiments, 32 dual-core virtual machines are deployed to run the `lammps` application in an 8-node cluster. The detailed configuration is presented in Section 3.1. Along with the parallel application, instances of `libquantum` run in a subset of nodes, as an interfering co-runner. The figure shows the normalized execution times when interference occurs by the co-running `libquantum` applications in 0-8 physical nodes.

To demonstrate the importance of interference modeling of parallel applications, the figure presents an expected performance with a *naive* proportional interference model. With the naive proportional model, the interference in a node affects the final execution time of the application proportionally. For example, if a task in a single node within an n-node cluster is slowed down due to interference, it affects the overall performance by a factor of $1/n$. The first set of bars shows the expected execution times of the application (`lammps`) with the naive model, and the second set of bars shows the real execution times of the same application. The x-axis represents the number of nodes where instances of `libquantum` are running, and thus `lammps` is experiencing interference.

As shown in the figure, the expected execution time with the naive proportional model increases linearly with the number of interfering nodes. However, the real runs exhibit completely different curves. Even when a single node suffers from interference, the execution time is significantly increased. Further increases of interfering nodes delay the execution with much slower rates. Such behavior of `lammps` is due to its parallelism and synchronization pattern. A slowdown in a single node will block the progress of the other nodes.

As shown by the motivation example, the performance of distributed applications cannot be simply estimated by the proportional aggregation of the performance of participating nodes. To address the problem, this paper proposes a modeling method for distributed applications under interference.

3. Modeling Interference

This section presents the interference sensitivity behaviors of distributed applications. First, we analyze the performance degradation when the number of nodes suffering from interference increases. Such *interference propagation* determines the performance of distributed applications, with a given number of nodes under the same interference intensity. Second, we investigate how heterogeneous intensities of interference in multiple nodes are reflected in the final performance of the applications.

Type	Name	Size	Abbrev.
SPEC MPI2007	104.milc	mref	M.milc
	107.leslie3d	mref	M.lesl
	113.GemsFDTD	mref	M.Gems
	126.lammps	mref	M.lmps
	132.zeusmp2	mref	M.zeus
	137.lu	mref	M.lu
NPB	cg	class D	N.cg
	mg	class D	N.mg
HADOOP	Kmeans	75 MB	H.KM
SPARK	PageRank	1M vertices with 12M edges	S.PR
	Collaborative Filtering	30 users on 100 movies	S.CF
	WordCount	4.2GB	S.WC
SPEC CPU2006	403.gcc	ref	C.gcc
	429.mcf	ref	C.mcf
	436.cactusADM	ref	C.cact
	450.soplex	ref	C.sopl
	462.libquantum	ref	C.libq
	483.xalancbmk	ref	C.xbmk

Table 1. Application configuration and data set

3.1 Methodology

For our study, we use a cluster composed of eight physical host nodes connected via a 10 Gigabit Ethernet switch. Each of the hosts is configured with two Intel Xeon Octa-core E5-2650 processors and 64 GB memory. The Xen hypervisor with version 4.3.2 is used, and the guest operating system for virtual machines (VMs) is a para-virtualized Linux of kernel version 3.2.59. For each VM, it is configured to have two virtual CPUs (vCPUs) with 5 GB memory. Since a host has 16 physical cores in total, the total 8 VMs can be placed on each of the hosts. Since we use compute-intensive workloads, the experimental setup does not over-commit the total vCPUs more than the available physical cores.

Table 1 shows application workloads and their input sizes used in our experiments. Four types of parallel workloads, SPECMPI2007, NPB, Hadoop, and Spark are used to investigate the interference model for distributed applications. Applications in SPECCPU2006, which are not parallel workloads, are used as possible batch-oriented co-running workloads in Section 5. For the experiments, each distributed application uses 32 dual-core VMs (i.e. 64 vCPUs in total) running on 8 physical nodes with 16 cores. For each workload, we restricted that four VMs which belong to the same application, are always executed together on a host node in experiments. Therefore, only up-to two applications can be co-located in a node, reducing the complexity of our analysis.

3.2 Interference Propagation

The first aspect of interference modeling for distributed applications is how the effect of interference occurring in a subset of nodes propagates to the execution of the whole ap-

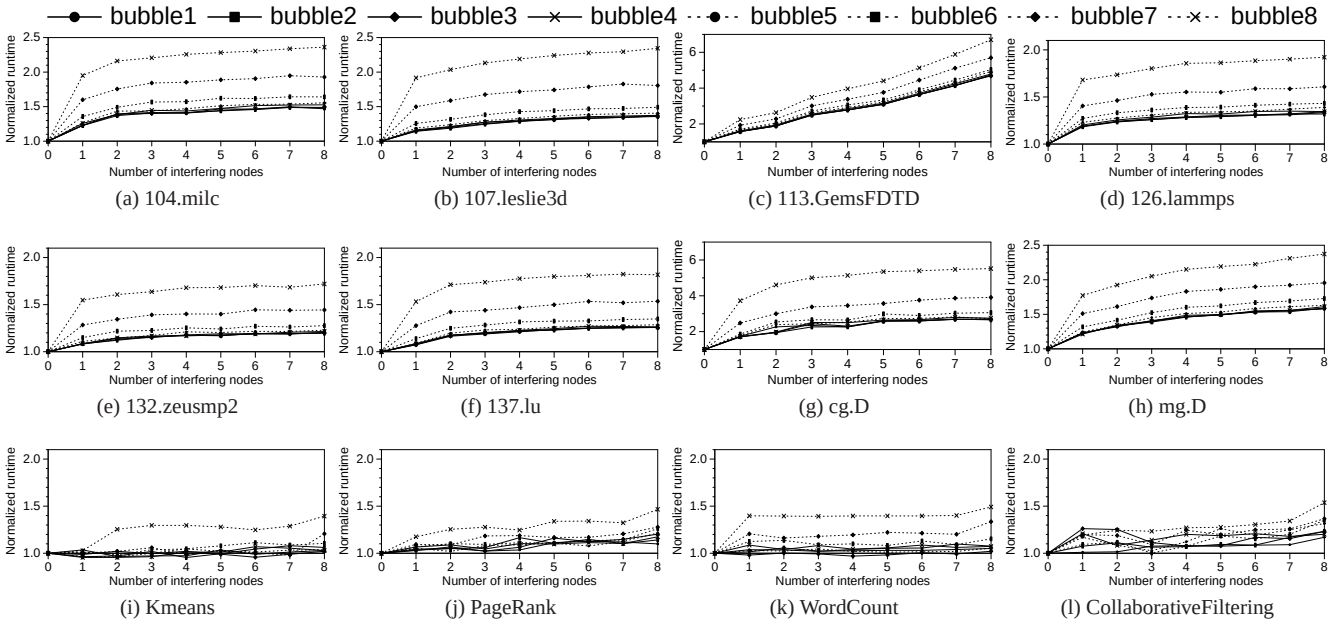


Figure 3. Execution time with varying bubble pressures with from 1 to 8 interfering nodes (normalized to no-interference)

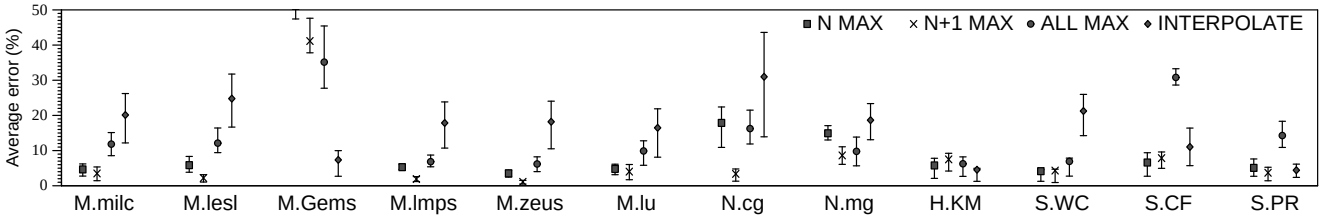


Figure 4. Average error by converting heterogeneous interference to homogeneous one with four policies on 8 hosts

plication spanning multiple nodes. In this section, we first present how the performances of our benchmark applications change when the number of physical nodes with interference (*interfering nodes*) increases. To simplify the analysis, all the interfering nodes suffer from the same interference intensity with the same bubble pressure. Figure 3 presents the normalized execution times when the number of interfering nodes increases from 1 to 8 physical nodes. The x-axis is the number of interfering nodes. Each graph shows 8 curves with different bubble pressures from 1 to 8 generated by a co-running bubble.

As shown in the figure, applications exhibit different interference propagation behaviors. In general, there are roughly three types of applications. In the first type, `high propagation`, the interference in one or two nodes affects the execution times significantly. The majority of applications, including `M.milc`, `M.lesl`, and `M.lmps`, exhibit such behaviors. The high propagation property becomes stronger with higher bubble pressures. In the second type, `proportional propagation`, such as `M.Gems`, the execution times increase proportionally to the number of interfering nodes. The behavior of this type of applications is similar to the collection of single-node applications. As more individual nodes are slowed down, the performance is proportionally degraded. Note that unlike the other MPI-based applications we used, `M.Gems` does not use any allreduce/allgather routines, and has much smaller numbers of barriers, having the low collective operations and showing linear increase patterns. The final type, `low propagation`, such as `H.KM` and `S.PR`, is relatively resilient to the interference, since it has low requirements for LLC capacity and memory bandwidth. Even if a co-running application monopolizes the shared resources, this type of applications is not affected significantly.

Various distributed workloads we examined exhibit different interference propagation behaviors. Furthermore, even in the same propagation type of applications, the performance curves quite differ, since the resilience to local interference varies by application behaviors. Even though application experts may know the general behavior of each application, and even its memory resource demand and usage pattern, it is extremely difficult to estimate the actual performance impact from the interference. Based on the observation, we conclude that per-application interference prop-

agation models are necessary to estimate the performance of each application, when some nodes have performance interference.

However, one critical problem of profiling and modeling the propagation behaviors of distributed applications is the number of profile runs to build the model. For single-node applications used by Mars et al. [13], it is necessary only to run each application with different levels of bubble pressure. For distributed applications running on multiple physical nodes, profiling runs must examine all possible numbers of interfering nodes, increasing the time complexity of the profiling run significantly. A naive method is to run with *N* different interference setups for an *N*-node cluster, and for each setup, different levels of bubble pressure must be examined. In Section 4, we will propose a scheme to reduce the required profiling runs drastically.

3.3 Interference Heterogeneity

The second difficulty of modeling interference for distributed applications is the heterogeneity of interference intensity in multiple interfering nodes. In the previous section, it is assumed that every interfering node suffers from the same level of interference. However, in real consolidated systems, each node may have a different interference level. If profiling runs must examine all possible intensities in different numbers of interfering nodes, the profiling space will become intractable. To resolve the explosion of the profiling space, we propose a method to convert the heterogeneous intensity in interfering nodes, to a homogeneous one. With this mapping, the model only needs to have the performance sensitivity curves with homogeneous intensity, as shown in Figure 3.

The rationale behind the feasibility of conversion process is that in some applications, the worst interference dominates the execution, while the average interference determines the overall interference level in other applications. In this work, we investigate and formulate four different mapping policies. First, `N max` policy considers only the number of interfering nodes under the worst interference intensity. For example, suppose there are four interfering nodes. If two of them suffer from the same high pressure, and the rest of two nodes are under lower pressures, the overall execution time of an application is similar to that of the same application with only two interfering nodes with the same high pressure. The effect of two interfering nodes with lower pressures is ignored. Second, `N+1 max` policy slightly augments `N max` policy. Instead of considering only the top intensity nodes, in this policy, the rest of interfering nodes are merged to an extra interfering node with the same top pressure, instead of ignoring them. For the same example of four interfering nodes, the heterogeneous intensity is mapped to a homogeneous intensity with 3 nodes with the same top pressure. The third policy, `all max`, assumes that the effect of the worst pressure even in a single node propagates directly to all the nodes. Therefore, it is mapped to a homogeneous intensity in which the worst pressure occurs in every node. The final policy, `interpolate`, uses an average intensity from all nodes as a representative interference in all the nodes. Note that in this work, we present a converting methodology to handle the heterogeneous intensity, and our methodology will be able to accommodate the addition of new policies, if any, for applications with entirely different behaviors.

Using the four different policies, we evaluate the behaviors of our workloads. However, examining all possible combinations of intensity levels from 1 to 8 with from 0 to 8 interfering nodes requires a large number of experimental runs which were impossible to test with our limited computing resources. With 8 physical systems, the total number of heterogeneous settings is 12,870. To find the best mapping policy, we use a sample-based approach. We randomly selected 60 heterogeneous interference configurations, and examined which of the four policies matches each application behavior.

Figure 4 presents the detailed error rates with the four different policies for each application. For each policy, an error bar with the minimum and maximum error rates is also presented. As shown in the figure, although not a single policy is the best for all applications, one or two policies for each application have very small error rates. Table 2 shows the best policy for each application with the average error, and standard deviation. As shown in the table, for distributed applications, at least, one of the policies can convert heterogeneous intensity to homogeneous intensity effectively with less than 9% average error and low standard deviation.

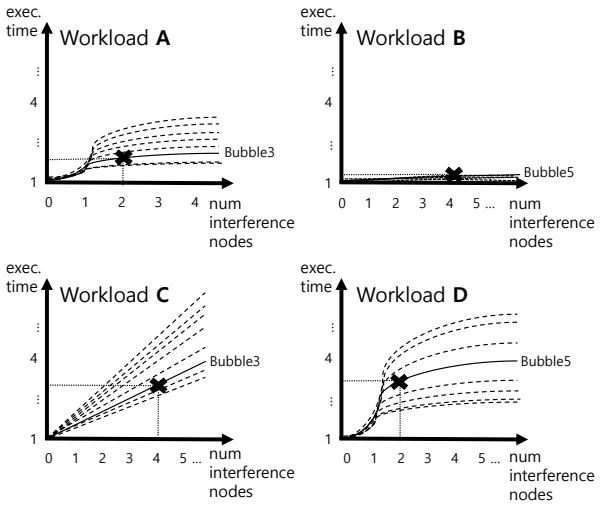
To find the best mapping policy, the profiling method must examine a statistically meaningful number of samples. For 12,870 total configurations, the 60 samples have a margin of error of about $+/-1.7$ on 99% confidence level, assuming the population distribution is normal and the standard deviation of the population follows the sample standard deviation. Due to the relatively low standard deviation as shown in Table 2, the confidence interval is short even on 99% confidence level. With a certain number of sample runs, the best mapping policy of each application can be found. Once the best policy is determined, only homogeneous intensity runs are necessary to build a model for the application.

3.4 Interference Model

To estimate the performance of distributed applications under diverse interferences in a subset of nodes, three modeling parameters must be obtained by profiling runs. First, the profiling process must measure the *bubble score* of each application, which is the interference intensity generated by the application. For MPI-based applications, the master participates in the computation like slaves. Thus, they generate a similar intensity of interference in all participating nodes, since each process exhibits a similar memory system behavior. However, for Hadoop and Spark applications, the master

Workload	Best policy	Avg. error(%)	Std. dev.
M.milc	N+1 MAX	3.50	2.18
M.lesl	N+1 MAX	2.20	1.60
M.Gems	INTERPOLATE	7.34	5.93
M.lmps	N+1 MAX	1.91	0.93
M.zeus	N+1 MAX	1.11	0.82
M.lu	N+1 MAX	4.01	2.35
N.cg	N+1 MAX	3.37	2.26
N.mg	N+1 MAX	8.62	3.16
H.KM	INTERPOLATE	4.55	1.41
S.WC	N MAX	4.15	7.95
S.CF	N MAX	6.60	4.96
S.PR	N+1 MAX	3.69	2.60

Table 2. The best heterogeneity mapping policy



Workload A,B,C,D's placement to 8 nodes

NODE 1	NODE 2	NODE 3	NODE 4	NODE 5	NODE 6	NODE 7	NODE 8
A	B	A	A	B	A	B	C
B	C	C	D	C	D	D	D

Workload	Bubble point	Bubble list	Converted bubble list
A (N+1 max)	5	[3, 2, 1, 1]	[3, 3, 0, 0]
B (ALL max)	3	[5, 2, 2, 1]	[5, 5, 5, 5]
C (INTERPOLATE)	2	[3, 5, 3, 1]	[3, 3, 3, 3]
D (N max)	1	[5, 5, 3, 2]	[5, 5, 0, 0]

Figure 5. Sensitivity profiles, bubble scores, and heterogeneity mapping policies

does not process any task, generating much lower interference. Thus, only among slaves, the interference levels are similar. In this work, we use the average interference intensity generated in all participating nodes as the bubble score of an application. However, further optimization is possible to consider the difference between the master and slaves for Hadoop and Spark applications. Second, to address the interference heterogeneity, the profiling process must find the per-application best mapping policy between heterogeneous and homogeneous interferences. Third, for each bubble pressure, the profiling runs must measure the interference sensitivity curves for increasing numbers of interfering nodes to represent interference propagation property as in Figure 3.

Algorithm 1 Binary-brute algorithm

```
1: var T: double[n][m + 1]; //initially T[i][j] = null
2:     t0: double;
3: t0 := measure(0, 0); //exec. time with no interference
4: for each i in 0..n − 1 do
5:     T[i][0] := 1; //norm. time with no interference
6:     T[i][m] := measure(i+1, m)/t0; //with max nodes at i+1
7:     T := profile_binary_row(T, i, 0, m, t0);
8:     T := interpolate_row(T, i);
9: end for
```

Figure 5 describes sensitivity curves and heterogeneity mapping policies for four workloads. The top table in the figure shows an example placement where workloads A, B, C, and D are placed in 8 nodes, with two different workloads sharing each node. In the bottom table, the workload column shows the best mapping policy for each workload, and the second column is the bubble scores of workloads. The third pressure list column is the heterogeneous interference actually occurring by the example placement, and the converted pressure list shows the homogeneous intensity converted by the best mapping policy of each application. With the information on the model, the performance of workloads A, B, C, and D can be estimated for the example placement. For example, workload A receives pressures of 3,2,1,1 with the placement. The heterogeneous interference can be converted to 3,3,0,0, since the best mapping policy is `N+1 max` for the workload. Then, using the curves for the pressure value of 3 with two interfering nodes, the performance of A under the interference is estimated.

4. Profiling Method and Model Validation

4.1 Interference Propagation Profiling

To profile the execution times of a distributed application under various interference settings efficiently, we devise two algorithms, *binary-brute* and *binary-optimized*. These algorithms selectively profile the execution times of a distributed application to construct the sensitivity curves with different bubble pressures and with different numbers of homogeneous interfering nodes as in Figure 3.

Each of the algorithms computes a matrix of execution times normalized to those under no interference for a distributed application. In an $n \times (m + 1)$ matrix, called T, where n is the total number of bubble pressures, and m is the total number of hosts, $T[i][j]$ is a normalized execution time when the number of interfering nodes with $i+1$ bubble pressure is j.

Algorithm *binary-brute* measures the execution of a target application for each different bubble level, which corresponds to each curve in Figure 3. For each curve, instead of examining all possible numbers of interfering nodes, it measures the execution times with a few selected numbers

Algorithm 2 Binary-optimized algorithm

1: **var** T: double[n][$m+1$]; //initially $T[i][j] = null$
2: t_0: double;
3: t_0 := measure(0, 0); //exec. time with no interference
4: $T[0][m]$:= measure(1, m)/t_0; //with max nodes at min bub.
5: $T[n-1][m]$:= measure(n, m)/t_0; //with max nodes at max bub.
6: **for** each i in $0..n-1$ **do**
7: $T[i][0] := 1$; //norm. time with no interference
8: **end for**
9: T := profile_binary_row(T, $n-1$, 0, m, t_0);
10: T := interpolate_row(T, $n-1$);
11: T := profile_binary_col(T, m, 0, $n-1$, t_0);
12: T := interpolate_col(T, m);
13: T := interpolate_all(T);

of interfering nodes (using function *measure*). The selection process uses a binary-search algorithm to eliminate unnecessary runs, and thus, if the performance difference with two different numbers of interfering nodes is small enough, then the profiling process does not need to run for the number of interfering nodes between the two setups (in function *profile_binary_row*). Once this profiling process is done, we predict each of unmeasured execution times by interpolating existing execution time values (in function *interpolate_row*).

Algorithm *binary-optimized* further reduces the profiling runs by exploiting the shape of curves is similar, regardless of bubble pressures. It constructs one curve with the highest bubble pressure with the same binary search as *binary-brute*. In addition to the top curve, it similarly measures or predicts the execution time with the maximum number of interfering nodes at different bubble pressures (in functions *profile_binary_col* and *interpolate_col*). By constructing the top curve and measuring the distance between curves among different pressure levels, the other curves with lower bubble pressures can be inferred, assuming their shapes do not change significantly. We can predict an unmeasured $T[i][j]$ by doing a sum by proportion based on the existing execution times (in function *interpolate_all*) as $T[i][j] := 1 + \frac{(T[i][m]-1)*(T[n-1][j]-1)}{T[n-1][m]-1}$.

4.2 Cost Reduction and Accuracy

We compare the performance of these two algorithms with random based algorithms, in terms of profiling cost and accuracy. The random-30% and random-50% algorithms randomly select 30% and 50% of all interference settings, respectively, and predict the execution times of other settings by interpolating the existing execution times. Note that in these algorithms, settings with no interference and with interference in all hosts for each bubble pressure are always selected and measured to construct sensitivity curves effectively.

Table 3 shows the accuracy of each of the four algorithms. The average profiling cost of an algorithm is computed as the average percentage of measured interference settings over all settings. The average error is computed as the average percentage of the difference between an estimated normalized execution time and a measured one for all settings. Figure 6 and Figure 7 present the average error and cost for each of the applications, respectively. In the results, for the binary-brute algorithm, it has the lowest error, but requires the highest cost among all the algorithms. For the binary-optimized algorithm, it shows a moderate error and requires the lowest cost. Considering both of profiling cost and accuracy, the binary-optimized algorithm can be used effectively to construct the sensitivity curves of an application, compared to the binary-brute and random based algorithms.

Prediction Algorithm	Average cost(%)	Average error(%)
binary-optimized	18.45	3.16
binary-brute	59.44	0.56
random-50%	49.23	5.31
random-30%	29.23	13.55

Table 3. Profiling cost and accuracy

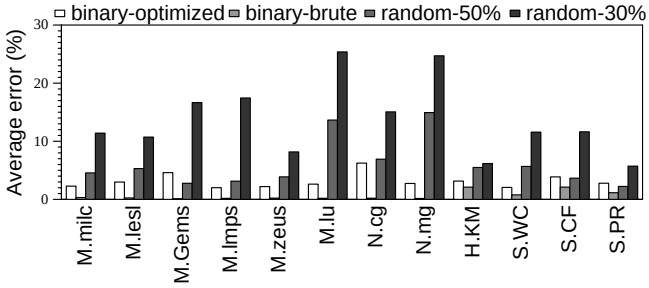


Figure 6. Prediction errors with four profiling techniques

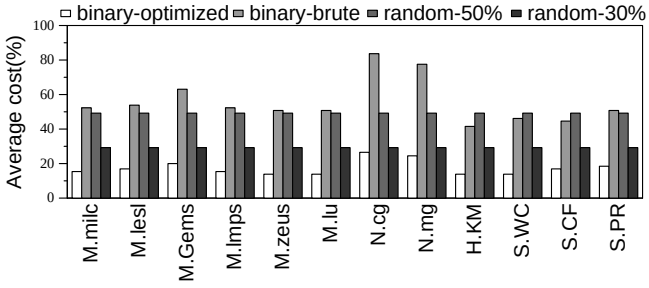


Figure 7. Profiling cost with four profiling techniques

Workload	Bubble	Workload	Bubble	Workload	Bubble
M.milc	4.3	M.lesl	3.9	M.Gems	2.4
M.lmps	1.0	M.zeus	1.4	M.lu	4.6
N.cg	3.9	N.mg	5.0	H.KM	0.2
S.WC	0.3	S.CF	0.5	S.PR	0.7
C.gcc	4.8	C.mcf	5.4	C.cact	3.8
C.sopl	4.9	C.libq	6.6	C.xbmk	4.3

Table 4. Bubble scores for the benchmark applications

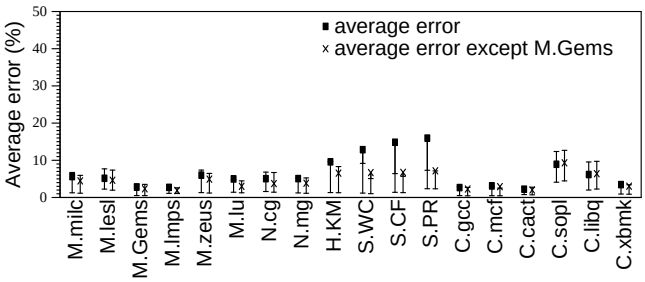


Figure 8. Average validation errors for applications

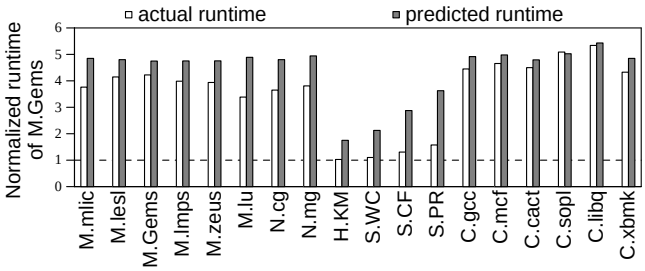


Figure 9. Validation errors with M.Gems

4.3 Model validation

We experimentally validate the interference-aware performance model by running two applications together in a cluster of multiple systems. We use the same system configuration as described in Section 3.1. For each application, the interference propagation model is constructed as in Figure 3, and the best interference heterogeneity mapping policy is also determined as in Table 2. Table 4 shows the measured bubble score for all applications, ranging from 0.2 to 6.6. They generate widely different intensities of interference.

Figure 8 presents the average error including 25% - 75% error bars for each application, when the application is co-running with all the applications (including itself). Most of workloads have less than 10% errors with the majority of them under 5%. The Spark applications have high error rates over 10%, but their average error rates are reduced significantly without the problematic co-runner, `M.Gems`. Figure 9 presents the predicted and actual runtimes of all applications running with `M.Gems`. `M.Gems` is one of the most unpredictable applications among our workloads, since its behavior is relatively unstable, depending on the co-running applications. It uniquely uses latency-sensitive blocked I/Os instead of common asynchronous I/Os. In Xen, when there are some idle CPU resources to dedicatedly execute Dom0 which handles network I/Os, the blocked I/O performance can be boosted. Since `M.Gems` is very sensitive to the availability of CPU for Dom0, our prediction accuracy is low for the Spark and Hadoop applications that have fluctuating CPU loads.

4.4 Limitations

Static Profiling: The first limitation of this study is its model construction method based on static profiling. This study assumes a priori knowledge of each application, and through separate profiling runs, the propagation and heterogeneity behaviors are measured before the application is used in production environments. The profiling is done only once for each application until the application binary or physical system configuration changes, and this study proposed to reduce the number of profiling runs with a binary search method and statistical sampling. Due to the static nature of model construction, the model may not reflect the dynamic changes of behavior accurately. For example, if an application has clear phase changes during its execution, and each phase has a different interference model behavior, the current single static model provides only an average behavior across different phases.

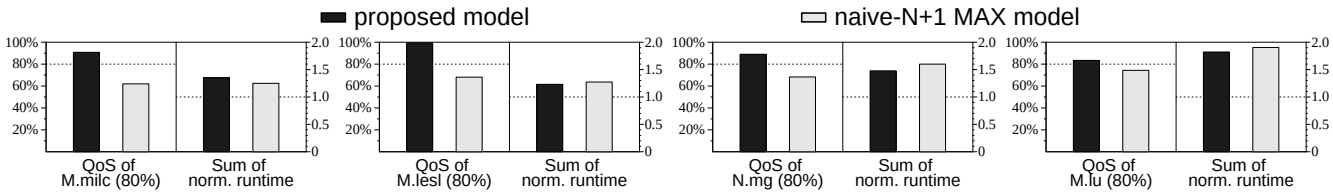
Pairwise Interaction: The second limitation of this study is its pairwise interaction within a node. In each node, only upto two different applications can be co-located to accurately model their interaction in the current modeling method. A possible solution for extending the model is to design a method to combine bubble pressures from multiple applications to a single score. Although the bubble design varies by resource types, in our bubble design for LLC and memory bandwidth, the dominant factor is LLC miss. In our scoring scheme, each score increase by 1 corresponds to the doubling of LLC misses. For example, when the same two bubble scores, *S*, are combined, the result score will be the sum of *S+1* and extra pressure by collision of the two combined applications. The extra pressure may be dependent on the original score, and this extra pressure estimation requires an additional modeling and validation effort.

Average Bubble Scoring: The proposed technique assumes that all the processes from an application across different nodes generate similar interference intensities, and uses a single average bubble score for the application. In the benchmark applications, the majority of MPI and Hadoop applications use the same program running on many nodes, and thus each process of the same application exhibits very similar LLC miss behaviors. Furthermore, as the balanced data distribution across processes attempts to equalize the amount of data handled by each process, the similarity among processes is very strong in our benchmark applications. One exception was the master node for certain applications, in which the master node performs significantly different tasks from the other worker nodes. A future extension of this study will be to support heterogeneous tasks within an application with potentially different bubble scores.

5. Case Studies

5.1 Placement Algorithm

When the interference propagation model and interference heterogeneity handling policy are profiled for each appli-



(a) M.zeus+C.libq+*M.mlic*+S.WC (b) C.xbmk+*M.lesl*+H.KM+N.mg (c) C.sopl+*N.mg*+S.CF+N.cg (d) *M.lu*+C.cact+M.lmps+M.Gems

Figure 10. QoS guarantee and runtimes normalized to solo runs

cation, a placement algorithm finds the best placement of applications to satisfy given constraints in a set of cluster nodes. Since the exhaustive search of all possible mappings is impossible, we develop an interference-aware VM placement algorithm based on simulated annealing [9]. A similar placement algorithm based on stochastic hill climbing has been studied for web-service workloads [12]. The placement algorithm attempts to find the global optimal point by locating a good approximation, although it is not guaranteed to find the optimal solution.

The algorithm starts by hypothetically mapping VMs randomly over hosts. To find the best VM placement under interference, it randomly selects two VMs running different workloads and swaps the locations of two VMs, if the new VM placement performs better while it satisfies given QoS constraints. The above process is repeated until the number of iterations reaches a predefined threshold. For each step with a hypothetical placement, the placement algorithm estimates the normalized execution time of each application using the proposed interference model. Note that this algorithm provides a reasonably good way to prove the effectiveness of our model as shown in the results below. However, other techniques rather than simulated annealing can be used, and they can also benefit from the interference model.

In this section, the experiments used the same system configuration described in Section 3.1. However, in these runs, each distributed application is running with 16 VMs, and four applications with total 64 VMs fully share the 8-node cluster with 16 cores per node. Since the current model supports only pairwise co-locations of different applications, our placement algorithms consider a set of 4 VMs running the same application in a host as a unit, instead of an individual VM (even though the algorithms in Sections 5.2 and 5.3 are described based on the unit of an individual VM). In this way, only one or two application workloads will be placed in a host. Since SPECCPU2006 applications are single-threaded applications, if a SPECCPU2006 application is chosen, 32 instances of the same application are running on 16 VMs, with 2 instances per VM, as a VM has two cores.

5.2 QoS-Aware Placement

An important use scenario is to support QoS guarantee for mission-critical distributed applications. When a mission-critical application is running, other less critical applications may or may not be allowed to share physical nodes. Without any interference model, it is impossible to predict whether the QoS of the mission-critical application will be satisfied or not. The QoS-aware placement algorithm is based on the aforementioned interference-aware placement algorithm with simulated annealing. After hypothetically distributing VMs for given workloads randomly over hosts in the system, the algorithm randomly selects two VMs running different workloads, and swaps the locations of the two VMs, if the new placement satisfies the delay constraint for the applications with the QoS requirement. In case that the current placement state already meets the delay constraint, it swaps randomly selected VMs only if the new VM placement still meets the constraint, and has a shorter total execution time. The above process is repeated for predefined iterations. The placement algorithm attempts to reduce the overall execution time while meeting the QoS constraint first.

Figure 10 presents the QoS support status and the sum of normalized runtimes for four application mixes. In the figure, one of the four applications for a mix (in italic) is a target application for QoS. The algorithm finds a placement to support the QoS for the application, and to maximize the overall performance. For the experiment, the QoS constraint is to guarantee 80% of the execution time compared to a solo run where the target application is running without any interference. For comparison, the result also shows a naive model for distributed applications. In the naive model, heterogeneous interference is converted to a homogeneous one with the `N+1 max` policy, which is the static best one, if we select a single policy for all the applications. Once the heterogeneity is eliminated, the overall performance is estimated proportionally by the number of interfering nodes (as the expected performance is estimated in Figure 2).

As shown in the figure, using the proposed model supports the QoS guarantee effectively within 80% of the solo run performance. However, using the naive model can violate the QoS as shown in the gray bar. The right side of the figures shows the sum of normalized runtimes. The execution time of each application is normalized to that in the

Index	Workload combination			
High perf. difference between best and worst (20%∼)				
HW1	N.mg	N.cg	H.KM	M.lmps
HW2	M.zeus	C.libq	H.KM	M.Gems
HW3	C.libq	N.cg	H.KM	S.PR
HM1	M.zeus	S.WC	M.Gems	S.PR
HM2	H.KM	M.Gems	M.lu	C.xbmk
HM3	S.CF	H.KM	M.Gems	M.Gems
Medium perf. difference between best and worst (5∼20%)				
MW	N.mg	H.KM	H.KM	M.lesl
MM	C.cact	C.libq	M.Gems	M.lmps
MB	N.cg	M.milc	C.libq	C.xbmk
Low perf. difference between best and worst (∼5%)				
L	M.lesl	M.zeus	M.zeus	N.mg

Table 5. Selected workload combinations

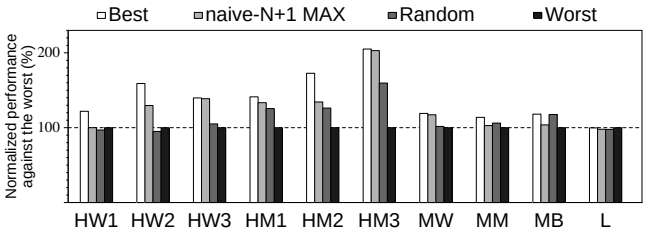


Figure 11. Normalized performance improvement

solo run, and the sum is weighted by the number of VMs used by each application. For example, 16 VMs of a SPEC-CPU2006 application have the same weight as the other distributed applications using the same number of VMs. For the mix (a), the proposed model increases the overall runtime slightly compared to the naive model to support QoS. For the rest of mixes, the overall runtimes are similar to or slightly shorter than those with the naive model.

5.3 Placement for Performance

Using the same placement algorithm as the QoS-aware placement, the second policy finds the placement for the best overall performance without any QoS constraints. For each application in a placement, its performance is defined as the speedup over the execution time of the same application in the worst placement. The overall combined performance from all the applications in a placement is the weighted average of the speedups from the applications, and the weight of each application is proportional to the number of VMs used by the application.

We chose 10 mixes with different behaviors as given in Table 5, to examine the effectiveness of our placement algorithm in real machines with a spectrum of mix characteristics. We ran each mix using three goals, *Best* maximizes the overall performance, *Worst* finds the worst performing placement for comparison, and *Random* shows the average results of five random placements. In addition, we also use the naive model explained in the previous section. In Table 5, the mixes are divided into high, medium, and low by the performance difference between the best and worst placements. The low difference mixes do not suffer from interference since generated interferences by applications are small or co-running applications are not very sensitive to interference. However, we still examine those mixes to show that the placement algorithm does not negatively affect such low difference cases either.

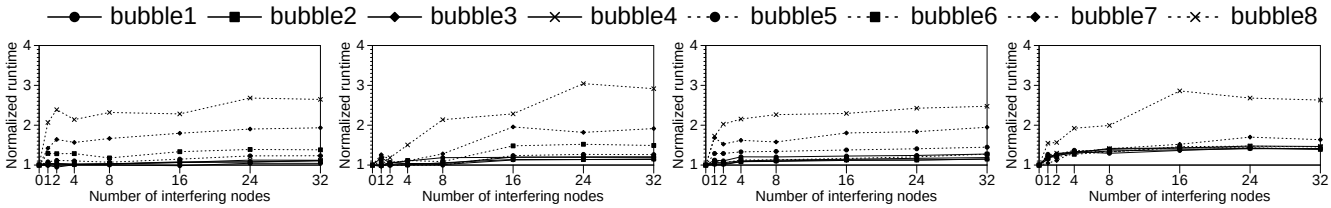
Figure 11 presents the average speedup of four applications for each mix. For the mixes with high difference, the placement algorithm quite effectively finds a good placement with a large performance improvement of up-to 105% for `HM3`. The average improvements with high and medium differences against the worst placement are 56.59% and 16.98%, respectively. For the naive model, the average improvements with high and medium differences are 39.86% and 7.94%, respectively. However, it shows unpredictable performance, which is often better or worse than the random placement. For all mixes, the best placement with our model exhibits the best performance. The result indicates that the proposed placement algorithm using the interference model effectively finds a good placement, and avoids the worst placement.

6. Results on Amazon EC2

To validate our modeling method for larger systems, we ran experiments with the Amazon EC2 service. For our experiments, we use 32 VM instances with *c4.2xlarge* type, which is configured with 8 vCPUs, 15 GiB memory, and high network performance. The interference model proposed in this paper requires to measure interference occurring in physical systems. However, in the public Amazon EC2 cloud, it is impossible to control the placement of VMs or to measure current interference from their co-running VMs. Thus, we co-locate applications within a VM provided by EC2. The experiments use four vCPUs to execute an application workload, while we reserve the remaining four vCPUs to execute bubble programs (or another application workload). The total number of vCPUs for a distributed application is 128.

There are two critical limitations of this EC2-based setup for our study. First, there is unmeasured interference from VMs owned by other users running on the same physical system. Second, the EC2 service may relocate VMs to different physical systems while they are not active. Therefore, it is very difficult to experiment on consistent physical systems with interference fully measured. To mitigate these uncontrollable effects, we selected four workloads which have relatively short execution times, to finish all the experiments before the system status changes. For application workloads, we use `M.milc`, `M.Gems`, `M.zeus` and `M.lu`.

For each of the four applications, the interference model is constructed with the proposed modeling method. Figure 12 shows the normalized execution times when the number of homogeneous interfering VMs (which execute bubble



(a) 104.milc (b) 113.GemsFDTD (c) 132.zeusmp2 (d) 137.lu

Figure 12. Execution time with varying bubble pressures and with from 1 to 32 interfering nodes on Amazon EC2

Workload	Best policy	Avg. error(%)	Std. dev.
M.milc	N+1 MAX	12.01	7.27
M.Gems	N+1 MAX	11.49	6.28
M.zeus	ALL MAX	6.40	4.52
M.lu	N MAX	5.28	4.36

Table 6. The best heterogeneity mapping policy on EC2

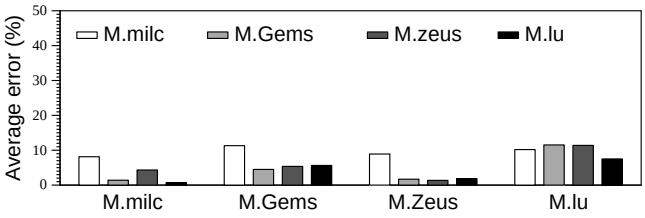


Figure 13. Validation errors for applications on EC2

programs along with the application workload) is 0, 1, 2, 4, 8, 16, 24 or 32. Table 6 shows the best mapping policy for each application with the average error and standard deviation. To find the interference heterogeneity model for each application, we randomly sample 100 heterogeneous interference settings, and measure the execution times on the selected settings. For each of the applications, the average error for the best mapping policy on Amazon EC2 is higher than that on our private cluster. A major reason is because we cannot control the placement of VMs and VMs by other users are likely running with our VMs on Amazon EC2.

Similar to Section 4.3, we run two applications together on 32 VMs to validate our model. Figure 13 shows the average performance estimation error for each application when it is executed together with all the other applications. For the benchmark workloads, the average errors are 3-10%.

The sensitivity curves, best mapping policy, and bubble score for an application are dependent on physical system configurations as well as the application behaviors. Therefore, separate interference-aware performance models need to be constructed for Amazon EC2. However, with the model parameters newly measured from the EC2 system, our model can still predict the performance under interference with modest errors for the larger scale systems.

7. Related Work

There have been earlier efforts on investigating techniques to reduce the effects of shared cache and NUMA. Zhuravlev et al. proposed scheduling algorithms in which threads are grouped and distributed among sockets in a multicore system such that the overall cache miss rate of the system is minimized [24]. A co-scheduling algorithm was proposed to group VMs with complementary resource demands [14]. In each time slice, it schedules tasks with different resource demands, reducing hotspots on the chip. To reduce shared cache impacts between threads (or VMs), low overhead cache partitioning techniques were studied [17, 19]. Nathuji et al. proposed to compensate the negative performance impact due to shared cache by provisioning additional resources for VMs running in a single system [15].

Besides cache contention, NUMA can complicate thread and VM scheduling and impact performance significantly. Blagodurov et al. developed a NUMA-aware scheduling algorithm that migrates the memory of threads to maintain NUMA affinity [3]. The effects of NUMA on shared memory and MPI-based applications have been investigated in virtualized environment [11]. To support NUMA systems, a commercial hypervisor, VMware ESX, provided optimization to migrate the memory of VMs [1], but shared LLC contention was not taken into account on scheduling. To minimize the effects of cache sharing and NUMA affinity for a cloud system, Ahn et al. proposed contention-aware scheduling techniques, which use live VM migration to schedule VMs dynamically for single-threaded workloads [2].

There have been several studies to reduce resource contention and balance load over physical hosts by using VM migrations. Wood at al. proposed techniques to monitor the resource usages of VMs, and initiate VM migrations to mitigate hotspots [20]. In VMware's Distributed Resource Scheduler (DRS) [10], it migrates running VMs for load balancing across physical hosts, based on the resource requirements specified by users. Similarly, VM migration is triggered when CPU utilization of a physical system becomes larger than some threshold for load balancing [6]. A decentralized migration technique was proposed to monitor network affinity between pairs of VMs, and migrate VMs to minimize communication overhead among VMs [18]. Scheduling techniques based on VM migration can be also

used for parallel workloads to reduce the latency caused by interference at runtime.

Performance prediction models can be used to schedule VMs efficiently. Zhu et al. proposed a power-aware consolidation algorithm for scientific workflows, which consolidates tasks with dissimilar resource requirements on the same node [23]. An interference-aware scheduling algorithm was studied for data-intensive applications in a virtualized data center, which characterizes the applications based on CPU and I/O resource usages [5]. In a task scheduling algorithm for MapReduce applications, it mitigates the effect of interference using an exponential interference prediction model, while improving data locality [4].

Several techniques to identify interference and predict the effect of interference have been investigated to improve performance of applications or resource utilization [13, 16, 21, 22], and also QoS-aware resource management techniques have been studied [7, 8]. CPI^2 detects interference based on CPI data and possibly throttles applications which affect the performance of other co-running applications badly [22]. Delimitrou and Kozyrakis investigated a QoS-aware scheduler that uses collaborative filtering techniques to classify an incoming workload regarding the effects of heterogeneous resources and interference [7], and developed a QoS-aware cluster management system which allows users to specify performance constraints (such as throughput) of applications, instead of resource reservation, and uses similar classification techniques for resource management [8]. The effects of heterogeneity caused by co-running jobs as well as different machine types have been analyzed for web-service applications in warehouse-scale computers, and a system (called Whare-Map) which maps jobs to heterogeneous machines using stochastic hill climbing technique has been proposed [12].

8. Conclusion

This study investigated how interference in individual nodes affects the overall performance of distributed parallel applications. Extending a prior *Bubble-Up* technique for modeling the interference effect in single-node applications, this work proposed a scheme to model interference propagation across multiple nodes and to address interference heterogeneity. Using the proposed performance model, the paper investigated two interference-aware placement algorithms, one for ensuring the QoS, and the other for maximizing the overall throughput of applications, as case studies. The proposed model can be used for the overall energy reduction to minimize the wasted CPU resources, when interference in some nodes is unavoidable for distributed applications with high interference propagation. A limitation of this study is, as one of the first studies for interference effect on distributed applications, that it relies on the profiling-based model construction for pairwise co-locations of the applications. Extending it to an online mechanism supporting co-location of multiple applications is our future work.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2013R1A2A2A01015514) and by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development)

References

[1] VMware ESX Server 2 NUMA Support. White paper. .

[2] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-ri Choi, and Jaehyuk Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[3] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2011.

[4] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013.

[5] Ron C. Chiang and H. Howie Huang. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[6] Hyung Won Choi, Hukeun Kwak, Andrew Sohn, and Kyusik Chung. Autonomous learning for efficient resource utilization of dynamic VM migration. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS)*, 2008.

[7] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[8] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[9] RW Eglese. Simulated annealing: a tool for operational research. *European journal of operational research*, 46(3):271–281, 1990.

[10] Ajay Gulati, Ganesha Shanmuganathan, Anne Holler, and Irfan Ahmad. Cloud-scale resource management: Challenges

and techniques. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[11] Jaeung Han, Jeongseob Ahn, Changdae Kim, Youngjin Kwon, Young-ri Choi, and Jaehyuk Huh. The effect of multi-core on HPC applications in virtualized systems. In *Proceedings of the 5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC)*, 2011.

[12] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[13] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[14] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

[15] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[16] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[17] Moinuddin K. Qureshi and Yale N. Patt. 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)*, 2006.

[18] Jason Sonnek, James Greensky, Robert Reutiman, and Abhishek Chandra. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In *Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP)*, 2010.

[19] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 2002.

[20] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI)*, 2007.

[21] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[22] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, 2013.

[23] Qian Zhu, Jiedan Zhu, and Gagan Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[24] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 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)*, 2010.