Interference Management for Distributed Parallel Applications in Consolidated Clusters

Jaeung Han
School of Computing, KAIST
juhan@calab.kaist.ac.kr

Young-ri Choi
School of Electrical and Computer Engineering, UNIST
ychoi@unist.ac.kr

Jaehyuk Huh
School of Computing, KAIST
jhhuh@kaist.ac.kr

Abstract

Consolidating multiple applications on a system can improve the resource utilization of data centers. However, such consolidation can adversely affect the performance of the applications due to interference caused by resource contention. Despite many prior studies on interference in single-node systems, the interference behaviors of distributed applications have not been investigated thoroughly. In distributed applications, a local interference in a node can affect the whole execution of the application spanning many nodes. This paper studies an interference modeling methodology for distributed applications to predict the performance under interference in consolidated clusters. This paper characterizes the effects of interference for various distributed applications over different interference settings, and analyzes how diverse interference intensities on the multiple nodes affect the performance. Using the proposed method, we develop an interference-aware placement algorithm based on simulated annealing, which can efficiently consolidate multiple distributed applications.

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 Distributed applications, interference model, placement algorithm, consolidated system, cloud computing

1. Introduction

An increasing number of cores in multi-core processors has been exacerbating contentions on shared architectural resources such as shared last-level caches 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 [6, 10, 12, 13, 18].

However, most of the prior work have isolated the effect of interference within a physical system. Their model assumes 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 [10, 18], the performance of a critical application running on a single physical system is protected from a co-running batch application. By estimating resource contention and its performance impact on the co-running applications, the studies have prevented batch applications from reducing the performance of the critical application lower than a threshold.

In clouds, however, many critical applications are multi-node distributed parallel applications which span multiple physical systems (nodes). In such distributed applications, the effect of interference in a physical node affects the final latency of parallel applications differently, depending on how the distributed parallelism is exploited. 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. 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 parallel applications may depends on how the different levels of interference manifest themselves in the parallel execution.

For such distributed applications in consolidated systems, this paper investigates how interference in a subset of nodes affect the final execution times of the applications, exploring techniques to model the interference effect in such distributed systems. Unlike prior studies which have been isolating interference estimation within a physical system boundary [10, 18], 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 for distributed applications. For some applications, a localized interference in a single node can lower the efficiency of many other nodes with no interference.

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 some 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. 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 behaviors of the application is modeled based on a small number of profiling runs. A similar approach has been used in the prior work [10, 18]. Mars et al. first used a profile-based model for the initial mechanism for interference management [10], and their subsequent work extended the profile-based approach to on-line estimation [18].

Based on the proposed modeling technique, this paper also proposes an interference-aware placement algorithm for distributed applications. Using the performance model, the algorithm based on a simulated annealing technique finds the best placement of multiple distributed applications on a cluster of physical nodes. In addition to simulated placement results to examine a large space of application combination, this paper shows the potential performance benefit of the interference-aware placement with real machine runs.

The main contributions of this paper are as follows.

- 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. Unlike prior studies limiting interference effects within the physical node boundary, the paper characterizes the interference propagation effect for distributed applications.
- 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.
- Using the proposed model, this paper explores an interference-aware placement algorithm for distributed applications to a cluster of physical nodes. By reflecting the interference propagation behaviors, the placement algorithm improves the overall throughput.

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. Section 5 proposes an interference-aware placement algorithm with simulated annealing, and shows the simulation and real machine results for the placement algorithm. Section 6 presents the related work and Section 7 concludes the paper.

2. Quantifying Interference

This section presents a prior technique proposed by Mars et al. to model the effect of interference for applications running on single-node systems [10]. Among various sources of performance interferences, shared last-level caches (LLC) and memory bandwidth have been dominantly contended resources for compute-intensive applications [10, 13, 18]. 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 I/O bandwidth, as discussed by the prior studies [10, 13, 18].

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*, measures how the performance of each application is degraded by different intensities of interference. Based on the performance changes, the technique generates interference response curves (*sensitivity profile*) for each application with profiling runs [10]. In addition to the per-application interference sensitivity profile, it also measures how much interference an 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 [10, 18].

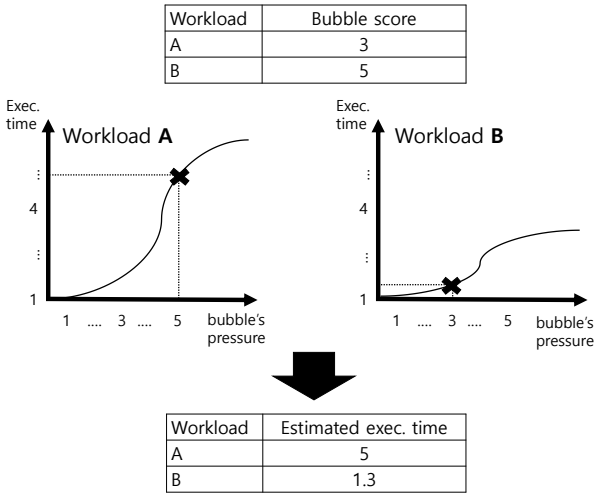


Figure 1. Interference sensitivity curves and bubble scores for two workloads

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 the profiling runs for each application under different interference levels, the scheme measures how much performance degradation each 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 by 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 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 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 [18]. 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 [13]. 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 a separate profiling run, 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.

It is not feasible to run large scale distributed applications in an isolated cluster for dynamic profiling runs as used by *DeepDive*. Therefore, 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.

3. Interference Model for Distributed Applications

In this section, we present 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 with the same interference intensity. Second, we investigate how heterogeneous intensity of interference in multiple nodes is reflected in the final performance of applications.

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 48 GB memory. In each host, 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 virtual machine (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 used in our experiments and their input sizes. Three types of parallel workloads, SPECMPI-2007, NPB, and Hadoop are used to investigate the interference model for parallel applications. Applications in SPECCPU2006, which are not parallel workloads, are used as possible batch-oriented co-running workloads in Section 5. For the experiments, in Section 3.2, each parallel workload consists of 32 VMs with 64 vCPUs in total, and in Section 3.3, it consists of 16 VMs with 32 vCPUs in total, to reduce the search space. For each workload,

Type	Name	size	abbrev.
SPECMPI2007	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	cg
	ep	class D	ep
	mg	class D	mg
	bt	class D	bt
	ft	class D	ft
	sp	class D	sp
HADOOP	Bayesian Classifier	8.0 GB	BC
	Kmeans	75 MB	KM
	WordCount	5.4GB	WC
SPECCPU2006	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

we restricted that four VMs which belong to the same application, are executed always together on a host node in both experiments. Therefore, only up-to two applications can be co-located in a node, reducing the complexity of our analysis. Note that `bt.D`, `ft.D`, `sp.D` are not used in the experiments with 16 VMs, since `bt.D` and `sp.D` cannot be configured with 32 processes, and `ft.D` cannot be executed due to lack of memory.

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 application execution spanning multiple nodes. Unlike single-node applications, in distributed applications, interference may occur only in a subset of 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 2 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. In the runs, each distributed application uses 32 dual-core VMs running on 8 physical nodes with 16 cores.

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 `milc`, `leslie3d`, and `lammps`, exhibit such behaviors. The high propagation property becomes stronger with higher bubble pressures. However, even in the applications with the same type, the performance curves quite differ, since the resilience to local interference varies by application behaviors. In the second type, `proportional propagation`, such as `GemsFDTD`, and `ft`, 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 of the distributed application is proportionally degraded. The final type of applications, `low propagation`, such as `ep` and `Kmeans`, is relatively resilient to the interference, since they have low re-

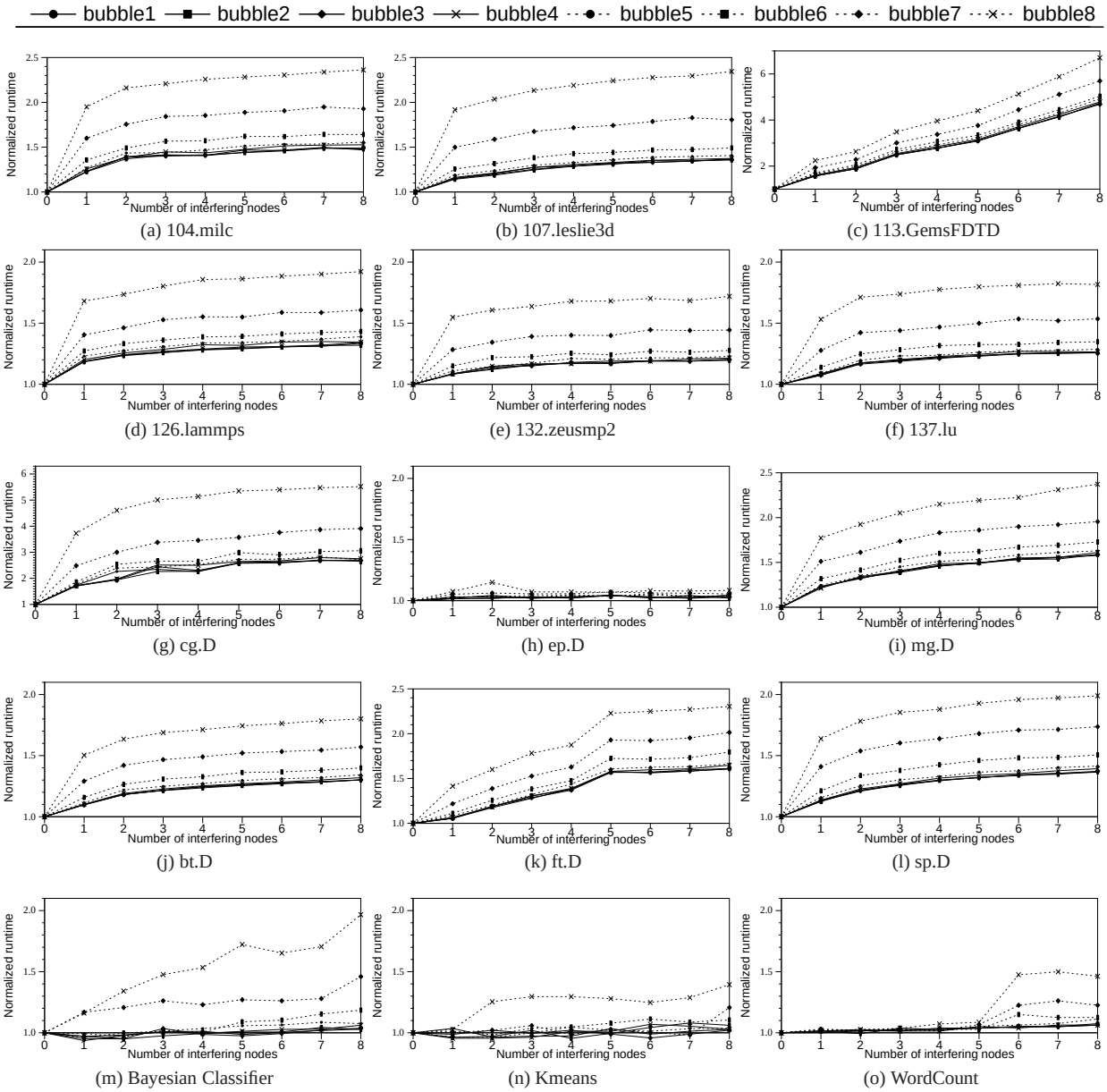


Figure 2. Execution time with varying bubble pressures and with from 1 to 8 interfering nodes (normalized to no-interference runs)

quirements for LLC capacity and memory bandwidth. Even if a co-running application monopolizes the shared resources, this type of applications are not affected significantly.

Various distributed workloads we examined exhibit different interference propagation behaviors. Furthermore, even in the same propagation type of applications, actual performance changes vary widely. Based on the observation, we conclude that per-application interference propagation 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. [10], 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. How-

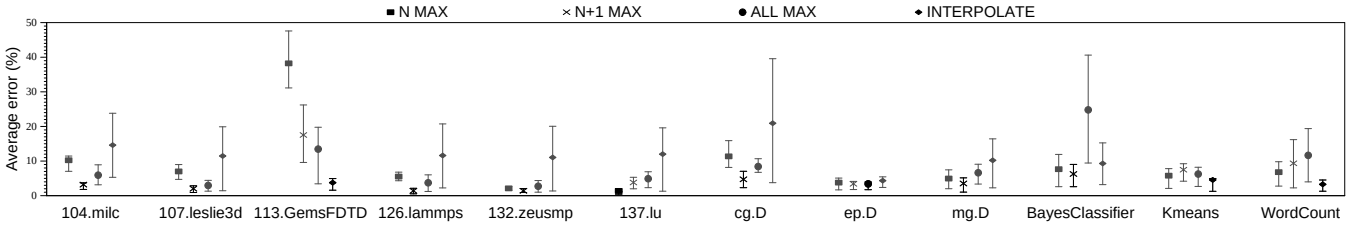


Figure 3. Average error by converting heterogeneous interference to homogeneous one with four policies

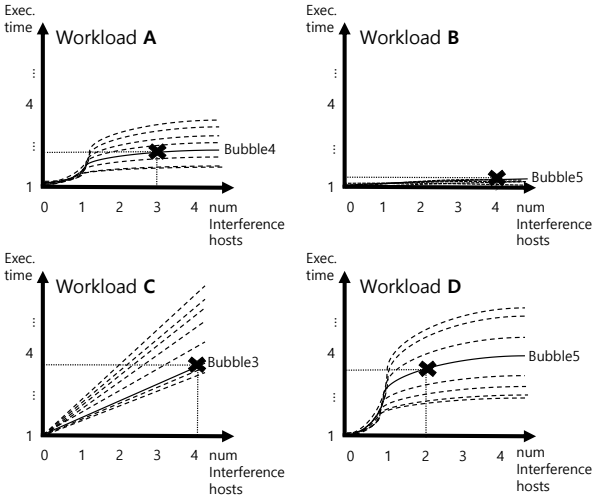
Workload	best policy	avg. Error(%)	std. dev
104.milc	N+1 MAX	3.12	2.19
107.leslie3d	N+1 MAX	2.02	1.50
113.GemsFDTD	INTERPOLATE	3.75	2.83
126.lammps	N+1 MAX	1.41	1.13
132.zeusmp	N+1 MAX	1.41	0.74
137.lu	N MAX	1.38	1.71
cg.D	N+1 MAX	4.72	2.98
ep.D	ALL MAX	3.41	2.96
mg.D	N+1 MAX	3.52	2.99
BayesClassifier	N+1 MAX	6.27	4.84
Kmeans	INTERPOLATE	4.55	5.41
WordCount	INTERPOLATE	3.28	2.63

Table 2. The best heterogeneous interference mapping policy

ever, in real consolidated systems, each node may have a different interference level. If the profiling runs must examine all possible intensities in different numbers of interfering nodes, the profiling space will become intractable. To resolve the explosion of 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 2. If a heterogeneous interference occurs by a placement, it is first converted to a homogeneous one, and then applied to the model.

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. To formulate the mapping policies, we investigate four different 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 a similar high pressure, and the rest two nodes are under lower pressures, the overall execution time of the 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 interfering nodes as a representative interference in all the nodes.

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 was impossible



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	C	C	A	C	A	D	D
B	B	A	D	B	D	B	D

Workload	Bubble score	Pressure list	Converted Pressure list
A (N+1 max)	5	[4, 4, 2, 1]	[4, 4, 4, 0]
B (ALL max)	4	[5, 2, 1, 2]	[5, 5, 5, 5]
C (INTERPOLATE)	2	[5, 4, 2, 1]	[3, 3, 3, 3]
D (N max)	1	[5, 5, 4, 1]	[5, 5, 0, 0]

Figure 4. Sensitivity profiles, bubble scores, and heterogeneity policies

to test with our limited computing resources. Instead, we random-sampled about 10% of the total space, and examined the four policies with the sampled configurations. In this analysis, we runs applications with the 16 VM setup with four physical nodes, to further reduce the search space. Figure 3 presents the more 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 has very small error rates. Table 2 shows the best policy for each application, 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 a 6% average error and low standard deviation.

This result indicates that the interference model does not require to examine every possible heterogeneous intensity levels in profiling runs. With a small number of 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. Distributed applications generates a similar intensity of interference in all participating nodes, since each process exhibits a similar memory system behavior. 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. The curves are essentially the same ones as shown in Figure 2.

Figure 4 describes the sensitivity curves and heterogeneity mapping policies for four workloads. A sensitivity profile for each workload must be generated with varying numbers of interfering nodes and varying pressures. The top table in the figure shows an example placement where workloads A, B, C, and D are placed in 8 nodes, with two VMs from two 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 fourth 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 those model information, the performance of workloads A, B, C, and D can be estimated for the example placement. For example, workload A receives pressures of 4,4,2,1 with the placement. The heterogeneous interference can be converted to 4,4,4,0, since the best mapping policy is `N+1 max` for the workload. Then, using the curves for the pressure value of 4 with three interfering nodes, the performance of A under the interference is estimated.

4. Profiling Method

This section describes how to reduce the required profiling runs to construct the sensitivity curves for distributed applications. As discussed in Section 3.3, the heterogeneity in interference intensity can be hidden by mapping it to a homogeneous interference execution. Therefore, to build the model, it is necessary to collect bubble sensitivity curves with different bubble pressures and with different numbers of interfering nodes.

4.1 Interference Propagation Profiling

To profile the executions time of a parallel application under various interference settings efficiently, we devise two algorithms, *binary-brute* and *binary-optimized*. These algorithms selectively profile the execution times of a parallel application to construct the sensitivity curves over various numbers of interfering nodes with homogeneous interference for each bubble pressure as in Figure 2.

binary-brute measures the execution of the target application for each different bubble level, which corresponds to each curve in Figure 2. For each curve, instead of examining all possible numbers of interfering nodes, it measures the execution times with a few selected numbers of interfering nodes, and the selection process uses a binary-search algorithm to eliminate unnecessary runs. If the performance difference with two different numbers of interfering nodes, is small enough, then the profiling process does not need to run the number of interfering nodes between the two setups.

binary-optimized further reduces the profiling runs by exploiting the shape of curves are similar, regardless of bubble pressures.

Algorithm 1 Binary-brute algorithm

```
1: var T: double[n][m + 1]; // initially T[i][j] = null
2:     t_0: double;
3: t_0 := profile(0, 0); // execution time with no interference
4: for each i in 0..n − 1 do
5:     T[i][0] := 1;
6:     T[i][m] := profile(i + 1, m)/t_0;
7:     T := profile_binary_row(T, i, 0, m, t_0);
8:     T := interpolate_row(T, i);
9: end for
```

Algorithm 2 Binary-optimized algorithm

```
1: var T: double[n][m + 1]; // initially T[i][j] = null
2:     t_0: double;
3: t_0 := profile(0, 0); // execution time with no interference
4: T[0][m] := profile(1, m)/t_0;
5: T[n − 1][m] := profile(n, m)/t_0;
6: for each i in 0..n − 1 do
7:     T[i][0] := 1;
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);
```

Algorithm 3 Function profile_binary_row(M, r, bc, ec, t)

```
1: if M[r][ec] − M[r][bc] > threshold_d then
2:     M[r][⌈(bc+ec)/2⌉] := profile(r + 1, ⌈(bc+ec)/2⌉)/t;
3:     M := profile_binary_row(M, r, bc, ⌈(bc+ec)/2⌉, t);
4:     M := profile_binary_row(M, r, ⌈(bc+ec)/2⌉, ec, t);
5: end if
6: return M;
```

Algorithm 4 Function interpolate_all(M)

```
1: for each i in 0..n − 2 do
2:     for each j in 1..m − 1 do
3:         M[i][j] := 1 + ((M[i][m]−1)∗(M[n−1][j]−1))/(M[n−1][m]−1)
4:     end for
5: end for
6: return M;
```

It constructs one curve with the highest bubble pressure with the same binary search as *binary-brute*. In addition to the top curve, it measures the execution time with the maximum number of interfering nodes at different bubble pressures. 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.

Each of our proposed algorithms computes a matrix of execution times normalized to those under no interference for a parallel application. The matrix of normalized execution times, called T in the binary-brute and binary-optimized algorithms, is an $n \times (m+1)$ matrix, where n is the total number of bubble pressures, and m is the total number of hosts. In the matrix, for each i-th row (where $0 \leq i < n$), a j-th element (where $0 \leq j \leq m$) is a normalized execution time of the application when the number of interfering nodes with $i + 1$ bubble pressure is j.

Prediction Algorithm	Average cost(%)	Average error(%)
binary-optimized	17.87	3.00
binary-brute	55.69	0.54
random-50%	49.23	5.05
random-30%	29.23	13.02

Table 3. Profiling cost and accuracy

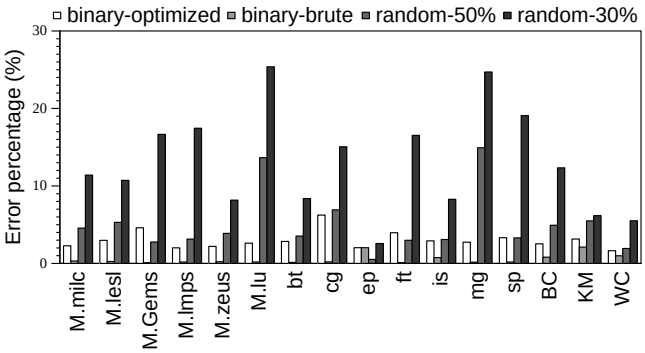


Figure 5. Prediction errors with four profiling techniques

In the binary-brute algorithm, we construct a sensitivity curve of an application for each bubble pressure. For each i-th row, we first need two (normalized) execution times with no interference (i.e. $T[i][0]$) and with the maximum number of interfering nodes at $i+1$ bubble pressure (i.e. $T[i][m]$). In the algorithms, function *profile(k,j)* returns a measured execution time when the total number of interfering nodes at k bubble pressure is j. In function *profile_binary_row*, we then profile the execution time with a setting where the number of interfering nodes becomes half, if the difference between $T[i][0]$ and $T[i][m]$ is larger than a predefined threshold. We continue to profile execution times selectively in the lower half and upper half portions of the curve in the same way. Once the above process is done, we predict each of execution times which have not been profiled by interpolating existing execution time values in function *interpolate_row(T,i)*.

In the binary-optimized algorithm, we attempt to reduce the profiling overhead further. We first need three execution times with no interference, with the maximum number of interfering nodes at the minimum bubble pressure (i.e. $T[0][m]$), and with the maximum number of interfering nodes at the maximum bubble pressure (i.e. $T[n-1][m]$). We then construct a sensitivity curve for the maximum bubble pressure as described in the binary-brute algorithm. In addition, we profile or predict execution times with the maximum number of interfering nodes at all the bubble pressures in function *profile_binary_col* and *interpolate_col*. Finally, we predict all other execution times, by doing a sum by proportion based on the existing execution times in function *interpolate_all*.

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, a setting with no interference and a setting for all hosts with interference of each bubble pressure are always selected and profiled 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 profiled interference settings over all settings when an algorithm is used to construct the sensitivity curves over all the applications. The average error is computed as the average percentage of the difference between an estimated normalized execution time and a profiled normalized execution time over the profiled normalized execution time for each interference setting. Figure 5 and Figure 6 present the average prediction error and profiling 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.

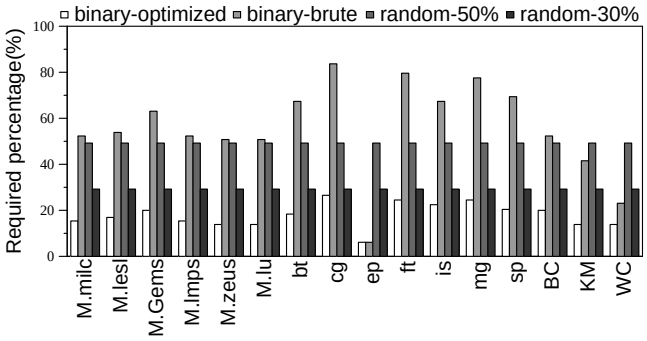


Figure 6. Profiling cost with four profiling techniques

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
cg	3.9	ep	3.8	mg	5.0
bt	4.8	ft	4.2	sp	5.6
BC	3.8	KM	0.2	WC	1.6
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 all applications

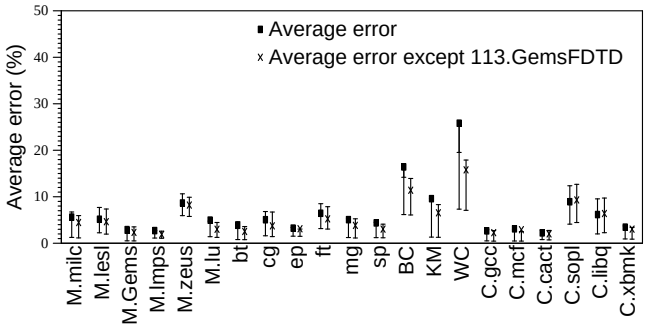


Figure 7. Average error for each application against all the other applications

4.2 Model Validation

In this section, 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, and each application uses 32 VMs sharing the 8-node cluster. For each application, the interference propagation model is constructed as discussed in the previous section, and the best interference heterogeneity policy is also determined as shown in Section 3.2. Table 4 shows the measured bubble score for

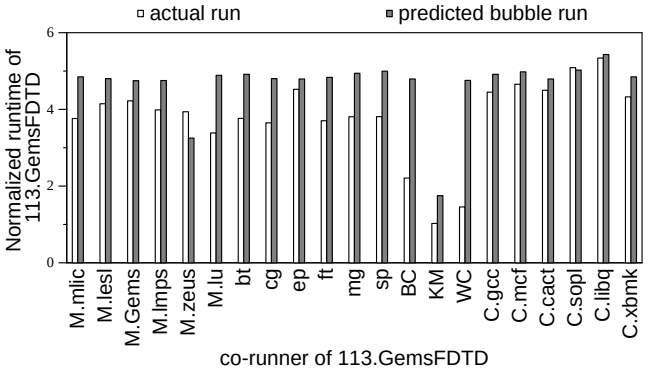


Figure 8. Validation error of 113.GemsFDTD running with the other workloads

all applications, ranging from 0.2 to 6.6. Our applications generate widely different intensities of interference.

Figure 7 presents the average error including 25% - 75% error bars for each application, when the application is co-running with all the other applications. As shown in the figure, most of workloads have less than 10% errors with the majority of them under 5%. There are two applications, `BC` and `WC`, with high error rates over 15%. However, for the two applications, the problematic co-runner is GemsFDTD. In the figure, the average error without GemsFDTD is also shown. Without GemsFDTD, the average error rates are reduced significantly. Figure 8 presents the predicted and actual runtimes of all applications running with `GemsFDTD`. GemsFDTD is the co-runner with the worst prediction rates when it is co-located with `BC` or `WC`. We observe that GemsFDTD is one of the most unpredictable applications among our workload applications, since its behavior is relatively unstable, depending on the co-running applications.

5. Case Study: Interference-Aware Placement

The interference-aware performance model for distributed applications can be used in several different ways. First, it can be used to predict performance for placing a newly submitted task to the available cluster nodes to maximize the overall throughput. Second, it can be used to maintain the delay restriction for distributed applications. If the delay constraint cannot be met, then migrating existing applications can also be possible on virtualized clusters as used by DeepDive [13]. Third, it can also be used for the overall energy reduction, minimizing the wasted CPU resources.

In this paper, as a case study for using the interference-aware performance model, we design a placement algorithm to maximize the overall throughput. When the interference propagation model and the interference heterogeneity handling policy is profiled for each application, the placement algorithm finds the best placement of applications to a given set of cluster nodes. This paper investigates only a static placement problem. In the static placement problem, there are multiple applications to schedule, and the total core count for all the applications is equal to the number of physical cores in the cluster. The placement algorithm must find the best mapping between a given set of applications and physical nodes to maximize the average of speedups from the applications. Although we investigate only the static placement problem as a case study applying the interference model, it can be extended to a dynamic placement problem combined with the VM migration mechanism as shown by Ahn et al. [2]. For example, when a new task is submitted, a similar placement algorithm will find the best mapping while migration of existing virtual machines is minimized.

Algorithm 5 VM placement algorithm

const k, // boltzmann constant
  $init\_t$, // initial temperature
  $cooling\_f$ // cooling factor, $0 < cooling\_f < 1$
input W_{list}: $< W_0, ..., W_{w-1} >$; // parallel worloads
  T_{list}: $< T_0, ..., T_{w-1} >$; // normalized exec. time matrix
  S_{list}: $< S_0, ..., S_{w-1} >$; // bubble scores
  H_{list}, H'_{list}: $< H_0, ..., H_{m-1} >$; // VM placement state of hosts, initially no VM on a host
var $temperature$, tot_time, new_time: double;
$temperature := init\_t$;
H_{list} := DistributeWorkloadsRandom(W_{list}, H_{list});
tot_time := GetTotalExecTime(W_{list}, T_{list}, S_{list}, H_{list});
while $temp > threshold_t$ **do**
  H'_{list} := SwapTwoVMsRandom(H_{list});
  new_time := GetTotalExecTime(W_{list},T_{list},S_{list},H'_{list});
  $\Delta := tot\_time - new\_time$;
  if $\exp(\frac{\Delta}{temperature \times k}) > random[0,1)$ **then**
    $tot\_time := new\_time$;
    $H_{list} := H'_{list}$;
  end if
  $temperature := temperature \times cooling\_f$;
end while

5.1 Placement Algorithm

Since the exhaustive search of all possible mapping is intractable, we develop an interference-aware VM placement algorithm based on simulated annealing [8]. The placement algorithm attempt to find the global optimal point by locating a good approximation.

For our algorithm, the following inputs need to be provided.

- W_{list}: a list of parallel workloads
- T_{list}: a list of normalized execution time matrices. T_i for every workload W_i in W_{list} is computed as discussed in Section 4.1.
- S_{list}: a list of bubble scores. S_i for every workload W_i in W_{list} is obtain as described in Section 3.4.
- H_{list}: VM placement state for each of hosts in the system. Initially no VM is placed on any of the hosts.

Algorithm 5 shows our VM placement algorithm. To place VMs for given workloads, we first distribute the VMs of the workloads randomly over hosts in the system in function *DistributeWorkloadsRandom*, and compute the total normalized execution time of the workloads for the VM placement state of the system (i.e H_{list}). In function *GetTotalExecTime*, we compute the the total (normalized) execution time of the workloads as the sum of the execution time of each workload. To compute the execution time of a workload W_i, the following steps are needed. First, for each host in which a VM running workload W_i is placed, we compute a bubble pressure that the VM experiences on the host, based on the bubble scores of the other co-running VMs. From the bubble pressures over the hosts, we can get a current interference setting of W_i, where the inference intensities may be heterogeneous over the host. Thus, we convert the heterogeneous interference setting to a homogeneous interference setting, based on the model discussed in Section XXX. Finally, we can get an estimated execution time of W_i from the execution time matrix with homogeneous interference (i.e. T_i).

To find the best VM placement under interference, we randomly select two VMs running different workloads, and swap the VMs in function *SwapTwoVMsRandom*. We then compute the total execution time of all the workloads for a new VM placement state (i.e. H'_{list}). If the difference Δ from the previous execution time to the new execution time is larger than zero, we can conclude that the

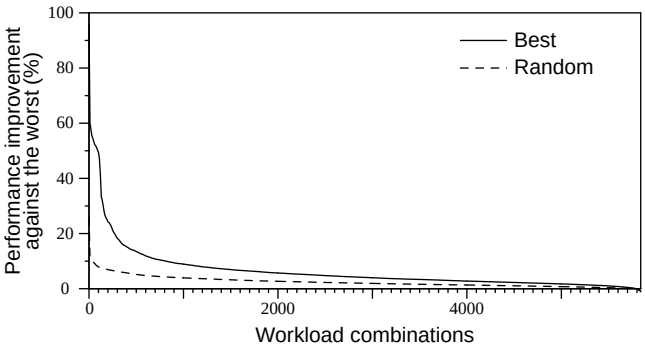


Figure 9. Performance improvement of best & random case compare with worst case each for all available workload combination with descending order

Random Trend	Index	Workload combination			
High performance difference between best and worst (20%∼)					
	HW1	C.libq	cg	cg	M.milc
	HW2	WC	mg	cg	KM
Worst	HW3	mg	cg	KM	M.lmps
	HW4	M.zeus	C.libq	KM	M.Gems
	HW5	C.cact	C.libq	cg	C.gcc
	HM1	WC	KM	C.mcf	M.Gems
	HM2	KM	KM	M.Gems	M.Gems
	HM3	KM	M.Gems	M.lu	C.xbmk
	HM4	C.libq	cg	M.lu	ep
Middle	HM5	KM	KM	M.Gems	ep
	HM6	M.zeus	cg	KM	C.mcf
	HM7	WC	cg	KM	C.gcc
	HM8	C.libq	WC	cg	ep
	HM9	C.libq	BC	cg	M.lmps
	HB1	C.cact	C.libq	WC	M.Gems
Best	HB2	BC	KM	M.Gems	M.Gems
	HB3	M.zeus	WC	M.lu	M.lu
Medium performance difference between best and worst (5∼20%)					
	MW1	mg.D	Kmeas	KM	M.lesl
	MW2	BC	KM	KM	M.lmps
Worst	MW3	M.zeus	mg.D	BC	KM
	MW4	C.sopl	cg.D	cg.D	M.milc
	MW5	C.sopl	BC	cg.D	C.xbmk
	MW6	C.cact	C.libq	M.Gems	M.lmps
Middle	MM1	M.zeus	C.libq	cg.D	M.lesl
	MB1	ep.D	mg.D	M.milc	M.lu
Best	MB2	C.sopl	KM	M.Gems	M.lmps
	MB3	M.zeus	ep.D	M.milc	M.milc
	MB4	cg.D	M.milc	C.libq	C.xbmk
Low performance difference between best and worst (∼5%)					
	L1	BC	M.lesl	cg.D	ep.D
	L2	M.zeus	BC	M.lu	M.lesl
None	L3	M.lesl	M.zeus	132.zeusmp	mg.D
	L4	WC	WC	ep.D	ep.D
	L5	cg.D	M.lu	M.Gems	M.lmps

Table 5. Selected workload combinations

new VM placement is better, and so we always move to the new state. If $\Delta \geq 0$, e^c where $c = \frac{\Delta}{temperature \times k}$ is always larger than 1. Note that if $\Delta = 0$, we also always more to the new state. If the move yields in a worse state (i.e. $\Delta < 0$), we probabilistically move to the new state. We then lower the temperature by applying the cooling factor, $cooling\_f$. The above process is repeated until the temperature becomes less then a predefined threshold.

5.2 Results

In this section, we show the performance benefit of the proposed placement algorithm with experimental results. We used the same system configuration described in Section 3.1, and in this result, each distributed application is running with 16 VMs. Four applications with 64 VMs fully share the 8-node cluster with 16 cores per node. Since examining many mixes in real machines is not possible, we first conduct a simulation-based result for the placement algorithm. The simulated analysis examines all possible 5820 mixes by randomly selecting four applications from Table 1. Since the SPEC application is a single-node application, if a SPEC application is chosen, 16 copies of the same application are used. The simulated placement estimates performance changes by different placements with the interference-aware performance model. The performance metric is the average speedup from all four applications. For each application, the performance is normalized to that with the worst placement (speedup), and the speedups of four applications are averaged. For SPEC applications, 16 VMs of the SPEC applications has the same weight as the other distributed applications using the same number of VMs.

Figure 9 presents the possible performance improvement with the best placement recommended by the placement algorithm. The x-axis represents all 5820 mixes from our applications, and the y-axis shows the performance improvement against the worst placement. Along with the best placement, the average random placement results from 5 random placements are also shown. In the simulated result, for 15% of possible mixes, the performance difference between the best and worst placement can be higher than 10%, and 94 mixes have more than 50% performance difference. The result conforms to the prior studies which showed that interference through the shared memory subsystem can often cause a significant performance degradation [2].

From the simulated results, we chose 33 mixes with various performance differences among the best, random, and worst placements, to examine the effectiveness of our placement algorithm in real machines with a spectrum of mix characteristics. Table 5 shows the mixes. To verify the placement algorithm for mixes with different behaviors, the mixes are divided into high, medium, and low by the performance difference between the best and worst placements. For high and medium mixes, the mixes are further divided by the performance difference between the best and random placements. The first column of the table shows whether the performance with the random placement is close to that with the best, middle, or worst placement. 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 10 presents the performance normalized to the worst placement, showing the average speedup of four applications with each bar. As shown in the figure, for the mixes with high differences, the placement algorithm quite effectively finds a good placement with a large performance improvement of up-to 104% for `HM-2`. Even if the random placement can perform effectively with very little extra performance gain compared to the best place (`HB mixes`), the placement policy selected by our scheme matches the random placement. The result indicates that the proposed placement algorithm using the interference model effectively finds a good placement, and avoids the worst placement.

5.3 Discussion

The static placement for throughput maximization we investigated in the previous section is the most straightforward application of the interference model. Another use scenario is to support a minimum

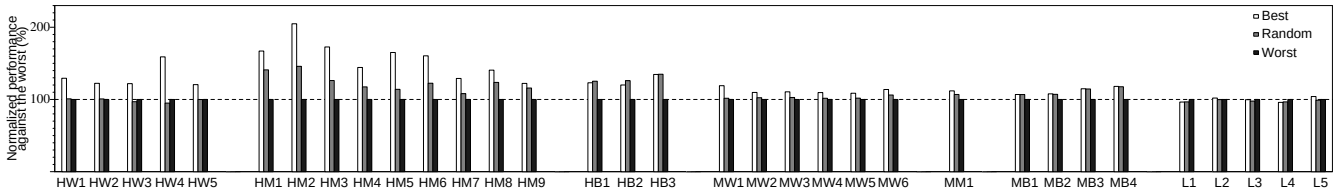


Figure 10. Normalized performance of best and random placement against the worst placement

performance guarantee for mission-critical distributed applications. When a mission-critical distributed application is running, other less critical applications may or may not be allowed to share the physical nodes. Without any interference model, it is impossible to predict whether the performance of the critical application will be degraded by less critical ones.

The second use scenario is for improving energy efficiency of the cluster. For distributed applications with high interference propagation, interference in a few nodes can slow down the whole application significantly. In such cases, the computing capability and power is wasted in un-interfering nodes. If interference in a subset of nodes is unavoidable, dynamic voltage and frequency setting (DVFS) can reduce the power state of un-interfering nodes to match the progress in the interfering node. Employing such energy management requires the interference propagation model proposed in this paper.

6. Related Work

There have been earlier efforts on investigating techniques to reduce the effects of shared cache and NUMA. In [21], 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. A co-scheduling algorithm was proposed to group VMs with complementary resource demands [11]. 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 as in [14, 16]. In [12], it was suggested to compensate the negative performance impact due to shared cache by provisioning additional resources for VMs running in a single system.

Besides cache contention, NUMA can complicate thread/VM scheduling and impact performance significantly. In [3], Blagodurov et al. developed a NUMA-aware scheduling algorithm that migrates the memory of threads to maintain NUMA affinity. To support NUMA systems, a commercial hypervisor, VMware ESX, provides 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 in [2].

There have been several studies to reduce resource contention and balance load over physical hosts by using VM migrations. In [17], Wood at al. proposed techniques to monitor the resource usage of CPU, memory, and I/O for all VMs in a data center, and initiate VM migrations to mitigate hotspots. In VMware's Distributed Resource Scheduler(DRS) [9], users can specify resource requirements of CPU and memory for each VM, and it migrates running VMs for load balancing across physical hosts. Similarly, VM migration is triggered when CPU utilization of a physical system becomes larger than some threshold for load balancing [5]. In [15], a decentralized migration technique was proposed to monitor network affinity between pairs of VMs, and migrate VMs to minimize communication overhead among VMs. Scheduling techniques based on VM migration can be also used for parallel workloads to reduce the latency caused by interference at runtime.

There are several efforts to schedule or map VMs based on performance prediction models. In [20], Zhu et al. proposed a power-aware consolidation algorithm for scientific workflows, which consolidate tasks with dissimilar resource requirements on the same node. In [4], an interference-aware scheduling algorithm for data-intensive applications in a virtualized data center was presented. Resource usages on CPU and I/O are used to characterize data-intensive applications.

Several techniques to identify interference and predict the effect of interference have been investigated to increase performance of applications or resource utilization [10, 13, 18, 19], and also QoS-aware resource management techniques have been studied [6, 7]. CPI^2 detects interference based on CPI data and possibly throttles applications which affect the performance of other co-running applications badly [19]. A QoS-aware scheduler that uses collaborative filtering techniques to classify an incoming workload regarding the effects of heterogeneous resources and interference was proposed in [6]. In a QoS-aware cluster management system [7], users are allowed to specify performance constraints (such as throughput) of applications, instead of resource reservation, and similar techniques as in [6] are used to classify applications for resource management.

7. Conclusion

The paper investigated how interference in individual nodes affects the overall performance of distributed parallel applications. Extending a prior *bubble* technique for modeling the interference in single-node applications, this work propsed a scheme to model interference propgation across multiple nodes and to address interference heterogenity. Using the proposed technique for interference-aware performance model, the paper investigated an interference-aware placement algorithm as a case study. A limitation of this study is, as one of the first studies for interference effect on distributed applications, it relies on the profile-based model construction. Extending it to an online mechanism is our future work.

References

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

[2] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[3] S. Blagodurov, S. Zhuravlev, D. Mohammad, and A. Fedorova. A case for numa-aware contention management on multicore processors. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[4] R. C. Chiang and H. H. 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 '11, pages 47:1–47:12, New York, NY, USA, 2011. ACM.

[5] H. W. Choi, H. Kwak, A. Sohn, and K. Chung. Autonomous learning for efficient resource utilization of dynamic vm migration. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 185–194, New York, NY, USA, 2008. ACM.

[6] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[7] C. Delimitrou and C. 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 '14, pages 127–144, New York, NY, USA, 2014. ACM.

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

[9] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad. Cloud-scale resource management: challenges and techniques. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, 2011.

[10] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. 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*, pages 248–259. ACM, 2011.

[11] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, 2010.

[12] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250, New York, NY, USA, 2010. ACM.

[13] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, pages 219–230. USENIX Association, 2013.

[14] M. K. Qureshi and Y. 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*, 2006.

[15] J. Sonnek, J. Greensky, R. Reutiman, and A. 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 '10, pages 228–237, Washington, DC, USA, 2010. IEEE Computer Society.

[16] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[17] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, 2007.

[18] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA'13: Proceedings of the 40th annual International Symposium on Computer Architecture*, volume 41, pages 607–618. ACM, 2013.

[19] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[20] Q. Zhu, J. Zhu, and G. Agrawal. Power-aware consolidation of scientific workflows in virtualized environments. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[21] S. Zhuravlev, S. Blagodurov, and A. 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*, 2010.