Algorithm Design for Performance Aware VM Consolidation

Alan Roytman
University of California, Los Angeles

Aman Kansal
Microsoft Research

Sriram Govindan
Microsoft Corporation

Jie Liu
Microsoft Research

Suman Nath
Microsoft Research

Abstract

Consolidation of multiple workloads, encapsulated in virtual machines (VMs), can significantly improve efficiency in cloud infrastructures. But consolidation also introduces contention in shared resources such as the memory hierarchy, leading to degraded VM performance. To avoid such degradation, the current practice is to not pack VMs tightly and leave a large fraction of server resource unused. This is wasteful. We present a performance preserving VM consolidation system that selectively places those VMs together that are likely to have the least contention. This drastically reduces the amount of wasted resources. While the problem of selecting the most suitable VM combinations is $\mathbf{NP}$-Complete, our system employs a practical method that performs provably close to the optimal. In some scenarios resource efficiency may trump performance and our system also implements techniques for this case. Experimental results show that the proposed system realizes over 30% savings in energy costs and up to 52% reduction in performance degradation compared to consolidation algorithms that do not consider degradation.

1 Introduction

Average server utilization in many data centers is low, estimated between 5% and 15% [9]. This is wasteful because an idle server often consumes more than 50% of its peak power [10], implying that servers at low utilization consume significantly more energy than fewer servers at high utilization. Additionally, low utilization implies a greater number of servers being used, resulting in wasted capital. One solution to prevent such wastage is to migrate applications to a public or private cloud and *consolidate* them on fewer servers.

The efficiency increase due to consolidation comes at a cost: a potential degradation in performance due to contention in shared resources. In particular, degradation due to shared caches and memory bandwidth is significant in magnitude as has been measured for a variety of real workloads [20, 4, 32, 18, 16, 2, 29, 15, 3]. Increases in execution time by up to several tens of percent are typical, though increases as high as $5\times$ have also been reported.

But performance is paramount for Internet services. Measurements on Amazon, Microsoft and Google services have shown that a fraction of a second increase in latency can result in losses as high as 1% to 20% of the revenue [12, 19, 24]. A knee-jerk reaction then is to forgo all or part of the savings from consolidation. In Google data centers for instance, workloads that are consolidated use only 50% of the processor cores [20]. Every other processor core is left unused simply to ensure that performance does not degrade.

We wish to perform consolidation in a manner that preserves performance, but *does not waste excessive resources in doing so.* The key intuition we rely on is that the degradation depends on which VMs are placed together. If we place together those VMs that interfere with each other minimally, then the excess resources left unused to overcome degradation would be minimized. The challenges in doing this are to (1) determine how much each VM will degrade when placed with each possible set of VMs, where that set may be taken from among all VMs to be consolidated, and (2) identify the sets that lead to the least overall degradation and use those in a performance preserving consolidation scheme. We design practical methods to address these challenges. The problem of determining the best sets of VMs to place together turns out to be $\mathbf{NP}$-Complete, and we design a computationally efficient algorithm that we prove performs close to the theoretical optimal. As a result, the excess resources left unused to preserve performance in our approach are significantly lower than in current practice.

An alternative approach to preserving performance after consolidation is to improve the isolation of resources in hardware [26, 4, 32, 2, 15], or software [1, 5, 25, 3, 29]. Further, excess resources may be allocated at run time [22] to overcome degradation. These approaches are complementary because they do not determine the least degrading VMs to be placed together in the first place. Our method can make that determination, and then these techniques can be applied, resulting in lower resource usage to preserve performance. Prior works have also proposed consolidation heuristics that attempt to reduce degradation [16] while packing VMs tightly to use all cores on active servers. We focus on preserving performance, and rather than proposing additional heuristics, we present a consolidation algorithm that provides a guarantee on how good the solution is compared to the optimal.

Specifically, we make the following contributions:

First, we present a *p*erformance *a*ware *c*onsolidation *man*ager, PACMan, that

minimizes resource cost, such as energy consumption or number of servers used. PACMan selects combinations of VMs that degrade the least when placed together. Since this problem is $\mathbf{NP}$-complete, PACMan uses an approximate but computationally efficient algorithm that is guaranteed to perform logarithmically close to the optimal.

Second, while interactive user-facing applications are performance constrained, batch data processing, such as Map-Reduce [7], may prioritize resource efficiency. For such situations we provide an "Eco" mode in PACMan that packs VMs tightly to fill all cores on servers that are used, but attempts to minimize the worst case degradation. We specifically consider worst case performance as opposed to average performance since, in Map-Reduce, the reduce stage cannot begin until all processes in the map stages have completed and hence, only the degradation of the worst hit map stage matters. We show that it is difficult to obtain efficient approximate solutions that have provable guarantees on closeness to the optimal. Hence, PACMan uses a suitable heuristic for this scenario.

Finally, we evaluate the design of PACMan for both the performance mode and Eco mode using actual degradation measured on SPEC CPU 2006 applications. For minimizing energy while preserving performance, PACMan operates within about 10% of the optimal, saves over 30% energy compared to consolidation schemes that do not account for interference, and improves total cost of operations by 22% compared to current practice. For the Eco mode, optimizing performance while using minimum possible resources, PACMan yields up to 52% reduction in degradation compared to naïve methods.

2 PACMan Design

This section describes the performance repercussions of consolidation and how our design addresses such issues.

2.1 Problem Description

The problem of performance degradation arises due to the following reason. Consolidation typically relies on virtual machines (VMs) for resource and fault isolation. Each VM is allocated a fixed share of the server's resources, such as a certain number of cores on a multi-core server, a certain fraction of the available memory, storage space, and so on. In theory, each VM should behave as if it is a separate server: software crashes or resource bottlenecks in one VM should not affect other VMs on the same server. In practice however, VM resource isolation is not perfect. Indeed, CPU cores or time slices, memory space, and disk space can be isolated

well using existing virtualization products, and methods have emerged for other resources such as network and storage bandwidth [21, 31]. However, there remain resources, such as *shared caches and memory bandwidth*, that are hard to isolate. Hence, consolidated workloads, even when encapsulated in VMs, may suffer resource contention or *interference*, and this leads to performance degradation. Such interference complicates VM consolidation, as shown in the following example.

Example: Consider a toy data center with 4 VMs, A, B, C, and D (Figure 1). On the left, the 4 VMs are placed on a single server each. Suppose the task inside each VM takes 1 hour to finish. The shaded portion of the vertical bars represents the energy used over an hour; the darker rectangle represents the energy used due to the server being powered on (idle power consumption) and the rectangles labeled with the VM name represent the additional energy consumed in VM execution (increase in server energy due to processor resource use). On the right, these VMs are consolidated on two servers (the other two are in sleep mode). The setup on the right is more efficient.

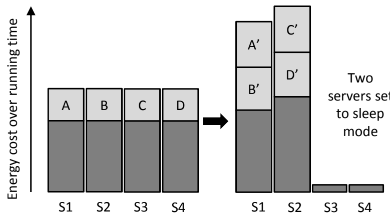

Figure 1: Energy cost change due to consolidation.

However, due to resource contention, the running time goes up for most of the jobs. Both the server idle energy and the additional energy used by each job increase due to the longer run time. The increase in energy consumption due to contention may wipe out some or all of the energy savings obtained by turning off two servers. Also, longer running time may violate quality of service (QoS) requirements.

The increase in run time depends on which jobs are placed together. For instance, in the above example, had we placed A and C together instead of A and B, the change in run times would have been different. Given a set of VMs and servers, there can be many possible consolidated placements (sets of VMs allocated to a server) differing in terms of which VMs share common servers. Each combination may lead to different performance degradation in the co-located VMs. Some servers may be allocated fewer jobs than others, leaving processor cores unused, to reduce the interference.

The choice of which VMs and how many VMs to place together on a server

yields a range of different operating points varying in *resource efficiency* and *performance*. One may minimize performance degradation by placing each VM in a separate server, but that obviously reduces efficiency. On the other hand, one may maximize efficiency by packing the VMs into the minimum number of servers required to satisfy the number of processor cores, memory and disk space requirements of each VM, but such packing hurts performance.

2.2 System Overview

Our key idea is to consider the extent to which different VMs are affected, determine the VMs that degrade the least when placed together, and then consolidate these VMs to the extent that performance constraints are not violated.

2.2.1 Assumptions

We make two assumptions about the system, described below.
VM to Processor Core Mapping: We assume that each VM is assigned one core, following closely the model developed in [18, 16, 2, 29, 15, 11]. The key reason for this assumption is that we are primarily concerned with resource interference due to shared cache hierarchy within a multi-processor chip. Such interference is the hardest to isolate using current virtualization technologies, and is well-characterized for VMs running on separate cores. Considering VMs that span multiple cores does not change the problem fundamentally. However, if multiple VMs share a single core, the nature of resource contention may change, and new characterizations would be needed.
Performance Degradation Data: We assume that the performance degradation suffered by each individual VM, when consolidated with any set of other VMs, can be determined using existing methods [18, 11, 20]. These methods can predict the degradation for multiple combinations of VMs based on an initial profiling of the individual VMs, without having to explicitly measure the degradation by running each combination. Explicit measurement may also be used for a small number of VMs, such as those used in [16]. This allows us to focus on the consolidation method itself.

2.2.2 Architecture

The PACMan system architecture is shown in Figure 2. The system consists of the following three components:

Conservatively Packed Servers: Customers submit VMs through appropriate cloud APIs. Ideally, a VM placement solution should determine the optimal placement for each VM as soon as it arrives, such that the entire set of VMs currently

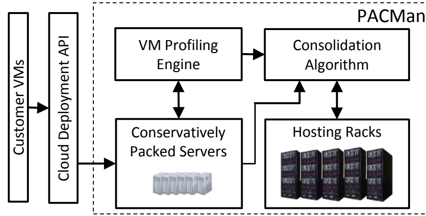

Figure 2: PACMan block diagram.

running in the cloud is optimally placed. However, since such an optimal online solution is not available, we focus on a *batched* operating scenario. The cloud operator collects the incoming VMs over a batching period (say 30 or 60 minutes) on these servers. The conservatively packed servers may comprise a small fraction (say 1%-5%) of the data center servers. The VMs are placed conservatively, i.e., wit significant resource left unused, on these servers to avoid interference (for example, by leaving several cores unused [20]). The use of these servers is transparent to the customer as their VM starts executing once placed accepted.

VM Profiling Engine: While a VM is running on the conservatively packed servers, profiling methods from [11, 20] are applied to the VMs. These methods can characterize a VM while it is running normally, and generate a set of parameters that allow estimating the performance degradation that will be *suffered* and *caused* by the VM when consolidated with other VMs. Their prediction accuracy is high (5-10% of actual performance), as measured on real data center and benchmark applications. For the PACMan prototype, we follow the method from [11]. Given n VMs and k core servers, this method requires only $O(n)$ measurements, even though the number of possible consolidated sets is $O(n^k)$. Each VM is mapped to a *clone* application, which closely mimics the application's interference signature. A discrete set of clones covers the entire spectrum of memory-subsystem interference behaviors. Thus, a potentially unbounded number of applications are mapped to a finite number of clones. A one-time profiling step maps a new VM to a known clone. The clones are then used as a proxy for predicting performance for different consolidation sets.

Consolidation Algorithm: At the end of each batching period, PACMan uses the VM consolidation algorithm proposed in this paper to place the VMs on *hosting racks* that comprise the bulk of the cloud's infrastructure. Most of the data center thus operates in an efficient manner using the near-optimal placement. The inputs to the consolidation algorithm are the interference characteristics for each of the accepted VMs, obtained by the profiling engine. These are used to determine the performance degradation that would be suffered if a set of VMs are placed together

on a server. Using this information, the algorithm carefully selects the sets of VMs to be placed together such that VMs with least degradation end up being on the same server, and leaves sufficient resources unused such that the performance constraints are not violated. Alternatively, if the number of servers available is fixed, the algorithm could attempt to minimize degradation. The detailed design of algorithms provided by PACMan is presented in the next two sections.

Temporal Demand Variations. Before discussing the algorithm, we note a practical issue that arises due to time-varying user demand for the VMs. The interference generated by a VM typically depends on the volume of demand being served by it. Interference generated is higher when serving peak demand than that at lower demand. Different applications may even peak at times that are inversely correlated, making them good candidates for consolidation. Therefore the performance degradation behavior must be profiled at each demand level. As a result the optimal placements will continuously change with varying demand. Both the requirements to profile at multiple demand levels and to continuously re-consolidate are impractical for realistic demand dynamics.

The scale-out design followed by many cloud-hosted applications offers a solution to this complication. Since the cost incurred by an application depends on the number of VM instances hosted in the cloud or a private data center, commercial products now offer solutions to dynamically scale the number of hosted VMs [23, 30]. Using this feature, hosted applications tune the number of active VMs to match the demand volume. As a result, each active VM operates at an efficient demand level. Hence we can safely assume that an active VM operates within a small band around its optimal demand level. Degradation is estimated at that demand level, and used in the consolidation algorithm. Changes in demand result in a varying number of active VMs, handled using the batched operation described above.

3 Performance Mode

The core of the PACMan consolidation system is the algorithm that determines the best sets of VMs to be placed together. In the first mode of operation, denoted the performance mode (P-mode), the consolidation algorithm determines the best sets and limits the sizes of the sets such that performance constraints are not violated. This may result in leaving some processor cores unused, unlike prior degradation-aware consolidation works that use up every core [16, 17].

To describe our algorithm precisely, and to establish its near-optimal solution quality, we abstract out the problem specification as follows.

Servers and VMs: Suppose m chip-multiprocessors (CMPs) are available, each with k cores. We are primarily interested in the inter-core interference within a CMP. The VMs placed on the same CMP suffer from this degradation. If a server happens to have multiple processor sockets, we assume there is no interference among those. As a result, multiple CMPs within a server may be treated independently of each other. We loosely refer to each CMP as a separate server as shown in Figure 3.

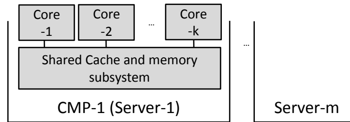

Figure 3: CMPs (referred to as servers) with k cores. VMs placed within the same server suffer degradation due to contention in the shared cache and memory hierarchy.

We are given n VMs to be placed on the above servers, such that each VM is assigned one core. Active servers may not be tightly packed but servers left completely unused can be turned off or repurposed.

Degradation: Suppose that the set of VMs placed together on a server are denoted by S. The choice of S influences the degradation suffered by each VM in that set. For singleton sets, i.e., a VM j running alone, there is no degradation and we denote this using a degradation $d_j = 1$. For larger sets, the degradation for VM $j \in S$ is denoted by $d_j^S \geq 1$. For example, for two co-located VMs, $S = \{A, B\}$, suppose A's running time increases by 50% when it runs with B, relative to when it runs alone, while B is unaffected by A. Then, $d_A^S = 1.5$ and $d_B^S = 1$.

We assume that adding more VMs to a set may only increase (or leave unchanged) the degradation of previously added VMs.

3.1 Consolidation Goal

The goal of consolidation is to minimize the resources used while preserving performance. This implies an upper bound on the allowable degradation due to interference suffered by each VM. We assume that the performance constraint is the same for all VMs though multiple quality of service classes, each with their own degradation limit, could be considered as well and do not fundamentally change the problem. The consolidation objective may be stated as follows:

P-Mode Objective: Minimize energy subject to a performance constraint:
Given

1. n VMs,

2. Servers with k cores,

3. VM degradations for all combinations of VMs placements with up to k co-located VMs,

4. A cost $w(S)$ for a set of VMs S placed on a server (e.g., energy cost), and

5. A degradation constraint $D \geq 1$ representing the maximum degradation tolerable for any VM.

Find a placement of the n VMs using some number, b, of servers, to minimize

$$\sum_{i=1}^{b} w(S_i)$$

where S_i represents the set of VMs placed on the ith server.

Cost Metric: The resource cost, $w(S)$, to be minimized may represent the most relevant cost to the system. For instance, if we wish to minimize the number of servers used, then we could use $w(S) = 1$ for any set S regardless of how many VMs S contains. The total resource cost would then simply be the number of servers used. To minimize energy, $w(S)$ could be defined as the sum of a fixed cost c_f and a dynamic cost c_d. Server idle power is often very high, exceeding 50% of the total power used with all cores utilized [10] and c_f models this. Capital expenses may also be added to the fixed cost c_f. The dynamic cost, c_d, models the change in energy with the number of VMs assigned to a server, and the energy overhead due to degradation, since higher degradation would lead to higher energy use over time (Figure 1). For concreteness, we consider the cost function $w(S)$ to be the energy cost. The specific values used for c_f and c_d are described in Section 5 along with the evaluations. Our solution is applicable to any cost function that monotonically increases with the number of VMs.

Batched Operation: The problem above is stated assuming all VMs to be placed are given upfront. In practice, following the setup from Figure 2, only the VMs that arrived in the most recent batching period will be available to be placed. Each batch will hence be placed optimally using P-mode consolidation, but the overall placement across multiple batches may be sub-optimal. Hence, once a day, such as during times of low demand, the placement solution can be jointly applied to all previously placed VMs, and the placement migrated to the jointly optimal placement. The joint placement satisfies the same performance constraints but may reduce resource cost even further.

The number of cores k is a fixed constant as it is fixed for a particular server deployment and is not to be optimized during placement. This ensures that the size

of the input is polynomial in n. In particular, if k were variable, the size of the input itself would be exponential in k, which would render the problem uninteresting.

3.2 Problem Complexity

It turns out that the complexity of the problem is different depending on whether the servers have only $k = 2$ cores or more than 2 cores.

Dual-Core servers: The polynomial-time algorithm for $k = 2$ proceeds as follows. We first construct an undirected, weighted graph on $2n$ nodes, n of which correspond to each of the n jobs, while the other n serve as "dummy" nodes. For each pair of job nodes $S = \{i, j\}$, if the performance of both jobs is acceptable when placed together ($d_i^S \leq D$ and $d_j^S \leq D$), we add an edge (i, j) with weight $w(\{i, j\})$. Recall that $w(S)$ represents the cost to run this set of two jobs on a server. For each job j, we associate a unique dummy node j' with j and add an edge (j, j') of weight $w(\{j\})$ (i.e., the cost to run job j alone on a server). Finally, we form a complete graph among the dummy nodes, so that for each pair of dummy nodes i' and j', we add an edge (i', j') of weight 0 (representing the cost of a non-existent or powered down server). Figure 4 illustrates such a graph for 4 jobs $\{i, j, k, l\}$ where set $\{k, l\}$ is not allowed due to performance degradation being greater than D on one of the two jobs, and the resource costs for other sets are marked on the edges.

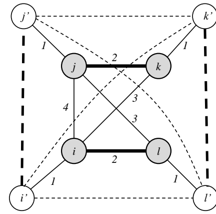

Figure 4: Nodes denote jobs and edge weights denote cost of a set (pair). Dashed lines show zero-weight edges. Bold edges show the minimum weight perfect matching.

We now observe that finding a minimum-weight perfect matching in this graph exactly corresponds to an optimal solution for the problem of minimizing resource cost. The minimum weight matching covers all nodes and has the minimum sum of edge costs, which is exactly the minimum cost server allocation. In the toy example illustrated in Figure 4, the bold edges represent a minimum weight perfect matching, and imply that placing the corresponding job pairs on servers leads to

the least cost (an edge connecting only dummy nodes need not be assigned a real server).

We add edges between all dummy nodes i' and j' to ensure that we can form a perfect matching (choosing such edges does not contribute anything to the cost, since these edges have weight 0). Note that the graph can be constructed in polynomial time, and it is well known that minimum-weight perfect matchings can be computed in polynomial time [16].

NP-Completeness: Present day servers however have more than 2 cores within a CMP. For servers with more than two cores ($k \geq 3$), the problem is $\mathbf{NP}$-Complete. This is because it can be thought of as a variant of the k-Set Cover problem. In the k-Set Cover problem, we have a universe U of elements to cover (each element could represent a VM), along with a collection C of subsets each of size at most k (the subsets could represent feasible sets of VMs with degradation below D). Placement of VMs on servers correspond to finding the minimum number of disjoint VM subsets that cover all VMs. Assuming $w(S) = 1$ for all sets S, the k-Set Cover problem becomes a special case of the P-Mode problem, i.e., solving the P-Mode problem enables solving the k-Set Cover problem. The k-Set Cover problem is $\mathbf{NP}$-Complete [8]. Hence, the P-Mode problem is $\mathbf{NP}$-Complete.

3.3 Consolidation Algorithm for Multiple Cores

Since the problem is $\mathbf{NP}$-Complete for $k \geq 3$ cores, we propose a computationally efficient algorithm that is near-optimal (i.e., our algorithm finds a placement of VMs such that the resultant resource use is provably close to optimal).

Using the profiling method described in Section 2.2, it is easy to filter out the sets that violate the degradation constraint. Among the remaining ones we want to choose those that have the least resource cost. Suppose the collection of allowed VM sets is denoted by $\mathcal{F}$.

First, for each allowed set $S \in \mathcal{F}$, the algorithm assigns a value $V(S) = w(S)/|S|$. Intuitively, this metric characterizes the cost of a set S of VMs. Sets with more VMs (larger set size, $|S|$) and low resource use ($w(S)$) yield low $V(S)$.

Second, the algorithm sorts these sets in ascending order by $V(S)$. Sets that appear earlier in the ascending order have lower cost and are favored.

The final step is to make a single pass through this sorted list, and include a set S as a placement in the consolidation output if and only if it is disjoint from all sets that have been chosen earlier. The algorithm stops after it has made a single pass through the list. The algorithm can stop earlier if all the VMs are included in the chosen sets. The first set in the sorted list will always be taken to be in the solution since nothing has been chosen before it and it is hence disjoint. If the second set is

disjoint from the first set, then the algorithm takes it in the solution. If the second set has at least one VM in common with the first, the algorithm moves onto the third set, and so on. The precise specification is given in Algorithm 1.

Algorithm 1 CONSOLIDATE($\mathcal{F}$, n, k, D)

1: Compute $V(S) \leftarrow \frac{w(S)}{|S|}$, for all $S \in \mathcal{F}$
2: $\mathcal{L} \leftarrow$ Sorted sets in $\mathcal{F}$ such that $V(S_i) \leq V(S_j)$ if $i \leq j$
3: $\mathbb{L} \leftarrow \phi$
4: **for** $i = 1$ to $|\mathcal{L}|$ **do**
5:     **if** S_i is disjoint from every set in $\mathbb{L}$ **then**
6:         $\mathbb{L} \leftarrow \mathbb{L} \cup \{S\}$
7: Return $\mathbb{L}$

Example: Consider a toy example with three VMs, A, B, and C and $k = 3$ cores. Suppose the characterization from the VM profiling engine results in the degradation numbers shown in Table 1. Suppose the performance constraint given is that no VM should degrade more than 10% ($D = 1.1$) compared to when running without consolidation and the cost metric $w(S)$ is just the number of servers for simplicity ($w(S) = 1$ for any set). A set with two VMs ($|S| = 2$) will have $V(S) = 1/2$ while a set with one VM will have $V(S) = 1$. Then filtering out the sets that cause any of the VMs to have a degradation greater than D, and computing the $V(S)$ metric for each set, we get the sorted list as: BC, AB, A, B, C (let us assume that running all three VMs together is infeasible due to performance degradation exceeding D). The algorithm first selects set BC and allocates it to a server (VMs B and C thus share a single server). The next set AB is not disjoint from BC and the algorithm moves to the subsequent set A. This is disjoint and is allocated to another server. All VMs are allocated and the algorithm stops.

VM Set	AB	AC	BC	A	B	C
d_{VM}^{Set}	$d_A = 1.1$ $d_B = 1.1$	$d_A = 1.0$ $d_C = 1.5$	$d_B = 1.0$ $d_C = 1.1$	1	1	1

Table 1: Degradations for VMs in the example.

Computation Complexity: The algorithm operates in polynomial time since sorting is a polynomial time operation, $O(|\mathcal{F}| \cdot log(|\mathcal{F}|))$. The subsequent operation requiring a single pass through the list has linear time complexity. At every step in the linear pass the algorithm needs to check if each VM in the set being selected has been assigned already and this can be achieved in constant time as follows. Maintain a boolean bit-vector for every VM indicating if it has been assigned yet. For the set being checked, just look up this array, which takes at most $O(k)$ time

per set since the set cannot have more than k VMs. Since the number of cores, k, is a constant, $O(k)$ is a constant. Also, after selecting a set we update the boolean array, which again takes constant time.

While the computation time is only polynomial in the size of the input, the size of the input can be large. The list of degradation values for all possible VM sets has size $|\mathcal{F}| = O(n^k)$ elements. While k is constant, n can be large for a cloud infrastructure hosting thousands of VMs. However, when the degradation estimation technique from [11] is used, all VMs are mapped to a clone and the number of clones does not grow with the number of VMs. We can treat all VMs that map to a common clone as one type of VM. The number of clones used to map all VMs then represents the distinct types of VMs in the input. For instance, for the characterization technique in [11], for quad-core servers, at most 128 types of clones are required, and not all of them may be used for a particular set of input VMs.

Suppose the n VMs can be classified into $\tau \leq 128$ types. Then, the algorithm only needs to consider all sets S from τ VM types with possibly repeated set elements. The number of these sets is $O(\tau^k)$, which is manageable in practice since τ does not grow very large, even when n is large.

The algorithm changes slightly to accommodate multiple VMs of each type. The assignment of value $V(S)$ and the sorting step proceed as before. However, when doing the single pass over the sorted list, when a disjoint set S is chosen, it is repeatedly allocated to servers as long as there is at least one unallocated instance of each VM type required for S. The resultant modification to Algorithm 1 is that $\mathcal{F}_\tau$ is provided as input instead of $\mathcal{F}$ where $\mathcal{F}_\tau$ denotes the collection of all feasible sets of VM types with repeated elements, and at step 5, instead of checking if the VMs are not previously allocated, one repeats this step while additional unallocated VMs of each type in the set remain.

Algorithm Correctness: It is easy to show that the algorithm yields a correct solution. The algorithm always assigns every VM to a server since all singleton sets are allowed and do appear in the sorted list (typically after the sets with large cardinality). Also, it never assigns a VM to more than one server since it only picks disjoint sets, or sets with unallocated VM instances when VM-types are used, while making the pass through the sorted list.

3.4 Solution Optimality

A salient feature of this algorithm is that the consolidation solution it generates is guaranteed to be near-optimal, in terms of the resources used.

Let ALG denote the allocated sets output by the proposed algorithm, and let OPT be the sets output by the optimal solution. Define the resource cost of the

proposed algorithm's solution to be $E(ALG)$, and that of the optimal algorithm as $E(OPT)$. We will show that for *every* possible collection of VMs to be consolidated,

$$E(ALG) \leq H_k \cdot E(OPT)$$

where H_k is the kth-Harmonic number. $H_k = \sum_{i=1}^{k} \frac{1}{i} \approx \ln(k)$.

In other words, the resource cost of the solution generated by the proposed algorithm is within $\ln(k)$ of the resource cost of the optimal solution. Given that k is constant for a data center and does not increase with the number of VMs, this is a very desirable accuracy guarantee.

Theorem 1. *For all inputs, the proposed algorithm outputs a solution that is within a factor $H_k \approx \ln(k)$ of the optimal solution.*

Proof. The proof is similar in spirit to the approximation quality proof for the weighted k-Set Cover problem [14, 6]. However, due to the difference that we cannot pick overlapping sets unlike in those works (since choosing sets in our setting corresponds to choosing a placement of VMs onto servers), a new proof is required.

By definition, we have

$$E(ALG) = \sum_{S \in ALG} w(S)$$

Assign a cost to each VM $c(j)$ as follows: whenever the proposed algorithm chooses a set S to be part of its solution, set the cost of each VM $j \in S$ to be $c(j) = w(S)/|S|$ (these costs are only for analysis purposes, the actual algorithm never uses $c(j)$). Hence,

$$E(ALG) = \sum_{S \in ALG} |S| \frac{w(S)}{|S|} = \sum_{S \in ALG} \sum_{j \in S} c(j) = \sum_{j=1}^{n} c(j)$$

where the last equality holds because the set of VMs in the solution is the same as all VMs given in the input. Then, since the optimal solution also assigns all VMs to servers:

$$E(ALG) = \sum_{j=1}^{n} c(j) = \sum_{S^* \in OPT} \sum_{j \in S^*} c(j)$$

where $S^* \in OPT$ is a set chosen by the optimal solution. Suppose, for the moment, we could prove that the last summation term above satisfies $\sum_{j \in S^*} c(j) \leq H_k w(S^*)$. Then we would have

$$E(ALG) \leq \sum_{S^* \in OPT} H_k w(S^*) = H_k \cdot E(OPT)$$

All we have left to prove is that, for any $S^* \in OPT$, we indeed have $\sum_{j \in S^*} c(j) \leq H_k w(S^*)$. Consider any set S^* in the optimal solution and order the VMs in the set according to the order in which the *proposed* algorithm covers the VMs, so that $S^* = \{j_s, j_{s-1}, \ldots, j_1\}$. Here, j_s is the first VM from S^* to be covered by the proposed algorithm, while j_1 is the last VM to be covered by the proposed algorithm in potentially different sets. In case the proposed algorithm chooses a set which covers several VMs from S^*, we just order these VMs arbitrarily.

Now, consider VM $j_i \in S^*$ immediately before the proposed algorithm covers it with a set T. At this time, there are at least i VMs which are not covered, namely $j_i, j_{i-1}, \ldots, j_1$. There could be more uncovered VMs in S^*, for instance, if the proposed algorithm chose set T such that T covers VMs $j_s, \ldots, j_i$, then all VMs in S^* would be considered uncovered immediately before set T is chosen. Moreover, since the optimal solution chose S^*, and since sets are closed under subsets, it must be the case that the proposed algorithm could have chosen the set $S = \{j_i, \ldots, j_1\}$ (since it is a feasible set and it is disjoint). At each step, since the proposed algorithm chooses the disjoint set T that minimizes $w(T)/|T|$, it must be the case that $w(T)/|T| \leq w(S)/|S|$. By our assumption that energy costs can only increase if VMs are added, we have $w(S) \leq w(S^*)$, and hence VM j_i is assigned a cost of $w(T)/|T| \leq w(S)/|S| \leq w(S^*)/|S| = w(S^*)/i$. Summing over all costs of VMs in S^*, we have

$$\sum_{j \in S^*} c(j) \leq \sum_{j_i \in S^*} w(S^*)/i = H_s \cdot w(S^*) \leq H_k \cdot w(S^*)$$

(since $|S^*| = s \leq k$). Hence, $\sum_{j \in S^*} c(j) \leq H_k \cdot w(S^*)$ indeed holds and this completes the proof. □

To summarize, if only dual-core servers are used, a polynomial time algorithm is available to find the optimal placements. However, the consolidation problem becomes $\mathbf{NP}$-Complete when the number of cores k is greater than 2, and we provide a polynomial time algorithm that is guaranteed to provide near-optimal solutions. Our algorithm performs within a logarithmic factor of the optimal, and this is asymptotically the best approximation factor guarantee one could hope for, due to the hardness of approximation lower bound that exists for the k-Set Cover problem [28].

4 Eco-Mode

In some cases, such as batch based data processing, resource efficiency may take precedence over performance. For such scenarios, PACMan provides a resource

efficient mode of operation, referred to as the Eco-mode. In this mode, the number of servers used is fixed and the VMs are tightly packed, using up every available core on the servers used. The goal is to minimize the degradation of the worst-hit VM. The worst case degradation is especially important for parallel computing scenarios where the end result is obtained only after all parallelized tasks complete and hence performance is bottle-necked by the worst hit VM.

The problem setup we consider is that n VMs are to be placed on a fixed number m of servers each with k cores (we have $n \leq mk$). We need to find the placement that minimizes the worst case degradation across all VMs. This is similar to problems previously considered in [16, 27, 17] in the sense that resources are constrained and performance degradation is to be optimized, though we are optimizing worst case performance instead of the average case. We now give a more formal description of the Eco-mode problem.

Eco-Mode Objective: Minimize Maximum Degradation:
Given

1. An existing placement of n VMs on m servers (VM sets $S_1, \ldots, S_m$), where each server has k cores, and
2. Degradations for all possible sets of VMs.

Find a placement of the n VMs to m servers to get a new placement $T_1, \ldots, T_m$ that minimizes

$$\max_{1 \leq i \leq m} \ell_i$$

where ℓ_i is the largest degradation suffered by any VM in set T_i.

4.1 Problem Complexity

As in the previous case, while the 2-core case can be solved in polynomial time, the Eco-mode problem becomes $\mathbf{NP}$-Complete for $k \geq 3$.

Dual-Core servers: We first describe how to solve this problem in the case when there are $n = 2m$ jobs, so that all m machines have exactly two jobs running on the two cores. We then explain how to generalize the problem to an arbitrary number of jobs $m < n \leq 2m$ (if $n \leq m$, this problem is trivial, since one can just assign each job to its own machine and the maximum degradation will be 1, which is optimal).

First suppose that the number of jobs is $n = 2m$, and consider all sets of jobs of size 2. Fix such a set, say $S = \{j_1, j_2\}$, and assign a value to this set $V(S) = \max(d_1^S, d_2^S)$, where d_1^S is the degradation of the first job in set S and d_2^S is the degradation of the second job in S. That is, $V(S)$ measures the largest

degradation in S. Now, sort all these size two sets of jobs according to the $V(S)$ values in ascending order (if multiple sets have the same value, ties may be broken arbitrarily). Sets in which both jobs experience no degradation (i.e., $d_1^S = d_2^S = 1$) will come earliest in the sorted order (if such sets exist), while the set containing the jobs which experience the largest degradation comes last in the sorted order. Suppose, for the moment, we could find the earliest point in this sorted order where we can choose m pairwise disjoint sets. That is, the first set S^* in the sorted order such that it is possible to choose m pairwise disjoint sets using only set S^* and sets that appear earlier in the sorted order. We claim that, if we find this point, then we have the optimal solution.

To see this, we first note that choosing m pairwise disjoint sets corresponds to finding a valid schedule, one in which all jobs are assigned to exactly one machine (since there are $n = 2m$ jobs, each of the m sets is size 2, and the m sets are pairwise disjoint). Now, we just have to argue optimality. We observe that the cost of our solution is precisely $V(S^*)$, since S^* contains the job with largest degradation in our solution and we set $V(S^*)$ to be the maximum of the two job degradations in S^*. Moreover, any solution which uses a set after S^* in the sorted ordering can only be worse than our solution, since any set T which appears after S^* has $V(T) \geq V(S^*)$ (since we sort in ascending order), and hence the cost of this other solution must be at least $V(T)$.

Suppose, towards a contradiction, that $V(S^*)$ is not the optimal solution. Consider the assignment of jobs to machines in the optimal solution. We observe that every machine has exactly two jobs, and all jobs are assigned to exactly one machine. Hence, this schedule corresponds to choosing m pairwise disjoint sets (each of size 2) in the sorted ordering. Let T be the last set in the sorted ordering chosen by the optimal solution. By definition of being optimal, we have $V(T) \leq V(S^*)$ (since the cost of the optimal solution is at most the cost of our algorithm's solution). If T appears after S^*, then we have $V(T) \geq V(S^*)$ (and hence, $V(T) = V(S^*)$) and the two solutions have equal cost. If $T = S^*$, then the two solutions again have equal cost. The only issue to take care of is when T appears earlier in the sorted ordering than S^* (in such a case, all we can conclude is $V(T) \leq V(S^*)$). However, we claim that this scenario cannot occur. Since the optimal solution corresponds to choosing m pairwise disjoint sets in the sorted ordering, and we find the earliest point S^* in the ordering when this is possible, it must be the case that either $T = S^*$ or T appears after S^* in the sorted list.

All we have left to argue, when $n = 2m$, is how to find the set S^* efficiently. After obtaining the sorted list, we construct a graph on n nodes, where each node corresponds to a job. For each set $S = \{j_1, j_2\}$ in sorted order, we add an edge to the graph between nodes j_1 and j_2. Each time we add an edge, we find a maximum cardinality matching in the graph. That is, a set of disjoint edges in the

graph that is maximum. The first set in the sorted list that causes the size of the maximum matching to reach m is precisely the earliest point in the ordering when it is possible to choose m pairwise disjoint sets. This clearly runs in polynomial time, since finding a maximum cardinality matching can be done efficiently. In fact, this algorithm can be made more efficient by performing a binary search on the sorted ordering. For a set S in the sorted ordering, if it is not possible to choose m pairwise disjoint sets using S and sets that appear earlier, then there is no point in checking sets to the left of S. On the other hand, if it is possible to choose m pairwise disjoint sets using S and sets that appear earlier, then there is no point in checking sets to the right of S.

In the case that $m < n \leq 2m$, the techniques are very similar. In this scenario, there is an optimal solution in which $n-m$ machines have been assigned two jobs, while the remaining $2m-n$ machines have been assigned a single job. With this in mind, we again sort a list of size 2 sets, and these sets get the same value as in the case when $n = 2m$. We again construct a graph on n nodes (corresponding to the n jobs), plus an additional $2m-n$ dummy nodes (these will correspond to the fact that we will have $2m-n$ machines which have been assigned one job). We connect each node corresponding to a job j to all $2m-n$ dummy nodes (this is in case job j should be run alone on a machine). We then proceed as before. That is, we go down the sorted list, adding an edge between jobs j_1 and j_2 for each set $S = \{j_1, j_2\}$. Each time we add an edge as we go down the list, we test whether the maximum cardinality matching is of size m. Similarly as before, this can be made more efficient by performing a binary search. Picking an edge between two job nodes (j_1, j_2) corresponds to assigning jobs j_1 and j_2 to the same machine, while picking an edge in which one of the endpoints is a job node while the other is a dummy node corresponds to assigning that job to a machine on its own.

Hence, we have an efficient, polynomial-time algorithm which solves this problem to optimality in the case of two cores for an arbitrary number of jobs.

4.2 Efficient Near-Optimal Algorithms

For the case of more than 2 cores ($k \geq 3$), it is not too difficult to show that this problem is $\mathbf{NP}$-Complete, and hence a polynomial time algorithm to compute the optimal solution is unlikely to be found (unless $\mathbf{P} = \mathbf{NP}$). The next question then is whether we can find an algorithm that computes a provably near-optimal solution.

Surprisingly, for $k \geq 3$ a computationally efficient algorithm with a reasonable guarantee on the solution quality cannot be found. So, not only is this problem $\mathbf{NP}$-Complete, but it is also $\mathbf{NP}$-Hard to approximate this problem to within any reasonable factor, including a factor that grows with the number of VMs. For

instance, a computationally efficient algorithm that can guarantee its solution to be within a factor n^{100} of the optimal cannot be found.

We accomplish this as follows. Consider a decision problem A that is $\mathbf{NP}$-Complete. By decision problem, we mean that, given an arbitrary input x for the $\mathbf{NP}$-Complete problem A, we must determine whether x is a "yes"-instance or a "no"-instance. Suppose we could come up with a reduction function f which maps inputs from problem A to inputs for another problem B which is a minimization problem (i.e., B is an optimization problem with an objective of minimizing some function). Moreover, suppose we can show that there exists a t such that if x is a "yes"-instance of problem A, then the optimal solution of $f(x)$ is at most t (i.e., $OPT(f(x)) \leq t$), and if x is a "no"-instance of problem A, then the optimal solution of $f(x)$ is at least $\alpha \cdot t$ (i.e., $OPT(f(x)) \geq \alpha \cdot t$). Note that, even though x is an instance of a decision problem, we have that $f(x)$ is an instance of problem B, so that it makes sense to consider its optimal solution. We claim that, if such a reduction function f exists, and if f runs in polynomial time, then approximating the problem B to within a factor less than α is impossible, unless $\mathbf{P} = \mathbf{NP}$.

To see this, we claim that if such a reduction function f exists, then we can correctly decide whether x is a "yes"-instance or a "no"-instance of problem A in polynomial time, which is a contradiction unless $\mathbf{P} = \mathbf{NP}$ (since A is assumed to be an $\mathbf{NP}$-Complete decision problem). Suppose, towards a contradiction, that we have an algorithm ALG with an approximation factor better than α for problem B. Given an input x for problem A, we compute $f(x)$ and check to see whether $ALG(f(x)) < \alpha \cdot t$. If $ALG(f(x)) < \alpha \cdot t$, then we output "yes", and if $ALG(f(x)) \geq \alpha \cdot t$, then we output "no." If $f(x)$ runs in polynomial time, we claim the procedure just described correctly decides whether x is a "yes"-instance or a "no"-instance in polynomial time. If x is a "yes"-instance, then $OPT(f(x)) \leq t$, which implies that $ALG(f(x)) < \alpha \cdot t$ and hence the procedure answers "yes" (note that we have $ALG(f(x)) < \alpha \cdot t$ since we assume that ALG guarantees an approximation ratio that is less than α). On the other hand, if x is a "no"-instance, then $OPT(f(x)) \geq \alpha \cdot t$, which implies that $ALG(f(x)) \geq \alpha \cdot t$ and hence the procedure answers "no" (note that $ALG(f(x)) \geq \alpha \cdot t$ since B is a minimization problem, and for such problems we always have $ALG \geq OPT$ for any input).

Thus, to achieve our inapproximability result for the Eco-mode problem, we must choose an $\mathbf{NP}$-Complete problem and prove that such a reduction function f exists. The decision problem of choice will be k-Dimensional Matching, and it is well known that this problem is $\mathbf{NP}$-Complete. In the k-Dimensional Matching problem, we are given as input k disjoint sets $X_1, \ldots, X_k$, each of size m, along with a set of k-tuples $T \subseteq X_1 \times X_2 \times \cdots \times X_k$. Here, $X_1 \times \cdots \times X_k$ is the set of all possible k-tuples, so we can think of each element $(a_1, \ldots, a_k) \in T$ as a compatibility constraint, where $a_i \in X_i$ for all i. We must decide if there exists a

subset of compatible k-tuples $M \subseteq T$ (i.e., a matching) such that any two k-tuples in the matching M are disjoint and $|M| = m$.

Given an input x for the k-Dimensional Matching problem, we will give a polynomial-time reduction function f with the property that if x is a "yes"-instance (i.e., there exists a k-Dimensional Matching), then $OPT(f(x)) \leq t$, and if x is a "no"-instance (i.e., there is no k-Dimensional Matching), then $OPT(f(x)) \geq \alpha \cdot t$ (in fact, in our proof we will show that $t = 1$ suffices).

Theorem 2. *For the Eco-mode consolidation problem, it is* $\mathbf{NP}$*-Hard to approximate the optimal solution to within any factor that is polynomial in the number of VMs and servers.*

Proof. The decision problem we will reduce from is the k-Dimensional Matching problem.

Suppose, towards a contradiction, that there exists a polynomial-time approximation algorithm for the Eco-mode problem with approximation ratio better than α. Consider an input x for the k-Dimensional Matching problem. We will construct an instance of the Eco-mode problem based on input x in polynomial time (this procedure will serve as our polynomial-time reduction function f). Recall that input x consists of k sets $X_1, \ldots, X_k$, each of size m, along with the set of compatible k-tuples T. Given this input, we construct m machines, each having k cores, and create $n = mk$ jobs.

All we have left to specify is how each job degrades in performance when it is assigned to a machine with an arbitrary set of other jobs. Observe that, since $n = mk$, every machine in any solution must have k jobs assigned to it, and hence we only need to describe how jobs in sets of size k degrade. Each element in $X_1 \cup \cdots \cup X_k$ can be thought of as a job, and choosing a k-tuple $(a_1, \ldots, a_k)$ (where each $a_i \in X_i$) to be part of the matching is equivalent to assigning the corresponding set of jobs to the same machine. With this in mind, for any set S of jobs of size k such that S corresponds to a compatible k-tuple in T, we degrade each job in set S by a factor of 1 (i.e., each job in S experiences no degradation in performance). For each set S that does not correspond to a compatible k-tuple in T, we degrade each job in S by a factor of α.

We now claim that if there is a k-Dimensional Matching, then $OPT(f(x)) \leq 1$. Suppose there exists a k-Dimensional Matching - that is, a matching M of size m. In particular, every element in $X_1 \cup \cdots \cup X_k$ appears in exactly one tuple since the tuples must be pairwise disjoint and $|M| = m$. Hence, there exists a solution for the Eco-mode problem of cost $t = 1$, where we assign the set of jobs corresponding to each k-tuple in M to the same machine. Note that, since the matching M consists of m compatible k-tuples, this implies that every job in the Eco-mode instance $f(x)$ experiences a degradation factor of 1 (since all jobs in a

set corresponding to a compatible tuple degrade by a factor of 1), and hence the optimal solution is at most 1 (in fact, it will be equal to 1).

On the other hand, suppose there does not exist a k-Dimensional Matching for the input x. We wish to show that for such inputs x, $OPT(f(x)) \geq \alpha \cdot 1 = \alpha$. Since there is no k-Dimensional Matching, this implies that in any assignment of jobs to machines, there must exist at least one machine which is assigned a set of jobs S that does not correspond to a compatible k-tuple (otherwise, there would be a k-Dimensional Matching). Hence, the set of jobs S already contributes a value of α to the objective function (since each job in S degrades by a factor of α). Since *any* assignment must incur at least this cost, the optimal solution must as well, and hence $OPT(f(x)) \geq \alpha \cdot 1 = \alpha$.

Thus, we have shown that approximating this problem to a factor better than α is not possible unless $\mathbf{P} = \mathbf{NP}$. In particular, we can set α to be any constant we desire. We can also set α to be any polynomial in n and m (in fact, α can be chosen to be any function in the number of jobs and machines, so long as the function is polynomial-time computable). □

The key implication of the theorem is that any computationally efficient consolidation algorithm designed for this scenario will have to rely on heuristics.

4.3 Eco-mode Algorithm

Since this problem is $\mathbf{NP}$-Complete, and based on Theorem 2 a polynomial-time algorithm that guarantees a good solution quality is also unlikely to be found unless $\mathbf{P} = \mathbf{NP}$, we must resort to heuristics. A heuristic algorithm may be based on some intuition about what method is likely to be beneficial. While such a consolidation algorithm may not be provably near-optimal, as long as it can improve the performance compared to naïve interference unaware methods currently in use, it will still be useful.

The algorithm we propose is perhaps very natural, and is based on a greedy heuristic. The key idea is to start with a random placement and greedily improve it using VM swaps. A swap refers to exchanging the placement of a VM with another. We limit the number of swaps allowed to terminate the search. Note that it is possible to reach *any* other placement (e.g., the optimal placement) by performing at most $G = (k-1)(m-1)$ swaps. This holds because for each server, we can imagine one of the VMs to be in the correct position on that server, and hence there can be at most $k-1$ VMs on that server that are out of place. By swapping two VMs, we can assign each VM which is on the wrong server to the right server. Hence, each server can be fixed in at most $k-1$ swaps. Once $m-1$ servers have been fixed, the last server must already be correct. However,

our heuristic is not guaranteed to find the optimal placement in $G = (k-1)(m-1)$ swaps, or any number of swaps.

The algorithm operates as follows. Start out with any initial placement of VMs. Now, consider all possible placements that are reachable from the existing placement in a single swap. In each such placement, for each server, compute the degradation of the worst hit VM on that server, using the degradation characterization from the VM profiling engine. Take the sum of these worst case degradations on all servers as the cost of that VM placement.

Among all possible placements reachable within a swap, greedily select the one with the lowest cost and actually perform the swap required to reach that placement. This uses up one allowed swap. Repeat the above process as long as there are additional allowed swaps available. This process stops when swaps no longer yield an improvement or the number of allowed swaps is exhausted.

For the Eco-mode, the algorithm above does not need all sets of VMs and their corresponding degradations up front. Rather, the degradations of just the reachable sets can be estimated on the fly, which helps reduce the size of the input.

5 Experimental Results

In this section, we evaluate the performance of PACMan. Ideally, we wish to compare the practical algorithm used in PACMan with the theoretical optimal, but the optimal is not feasible to compute (these problems are $\mathbf{NP}$-Complete) except for very small input sizes. Hence, we illustrate the performance of the proposed methods with respect to the optimal for a few small input instances ($n = 16$ VMs, $m \geq \lceil n/k \rceil$). For more realistic inputs, of the same order of magnitude as the number of VMs in data centers (10^3 VMs), we compare the performance to naïve methods that are either unaware of the performance degradation and with one current practice that leaves alternate processor cores unused [20]. For these cases, we also compute the degradation overhead compared to a hypothetical case where resource contention does not cause any degradation. This comparison shows an upper bound on how much further improvement one could hope to make over the PACMan methods.

5.1 Experimental Setup

The following server and workload characteristics are used for our experiments.

Degradation Data: We use measured degradation data for SPEC CPU 2006 benchmark applications. These degradations are in the same range as measured for Google's data center workloads in [20], and may hence be considered representa-

Application VMs	Degradations (%)
lbm, soplex	2, 19.7
soplex, soplex	10, 10
lbm, soplex, sjeng	2, 10, 4.1
lbm, povray, lbm	19.6, 5.32, 19.6
lbm, soplex, soplex, sjeng	14.56, 36.9, 36.9, 5.83
lbm, lbm, lbm, lbm	104.6 (each)

Table 2: Sample degradation data for the application VMs used in experiments. Degradations are measured on a quad-core processor. For combinations with only 2 or 3 VMs, the remaining cores are unused. Degradations over 100% imply that the execution time of the workload increases by more than twice.

tive of at least some real world workloads. In particular we select 4 of the SPEC CPU benchmark applications for which we have detailed interference data for all possible combinations: `lbm`, `soplex`, `povray`, and `sjeng` (some combinations shown in Table 2). These span a range of interference values from low to high. When experimenting with n VMs, we generate an equal number, $n/4$, of each.

Cloud Configuration: We assume that each server has $k = 4$ cores since quad-core servers are commonly in use, though $k = 2$ and $k = 6$ are also possible. While a server may have many cores across multiple processor sockets, the relevant value of k is the number of cores sharing the same cache hierarchy, since that is where most of the interference occurs.

5.2 Performance Mode

For the P-mode, a degradation constraint is specified and resource cost is optimized. The evaluation metric of interest is thus the resource cost. We choose energy as our resource metric. Each server has a fixed and dynamic energy component (Section 2), resulting in an energy cost $w(S) = c_f + \sum_{j \in S} d_j^S$. Here, the additional cost of each VM is being modeled as d_j^S. Considering that running 4 VMs each with an incremental cost of 1 or more would add an additional 4 units of dynamic resource cost, we set the fixed cost $c_f = 4$ to reflect about 50% of the total server energy as the fixed idle cost, which is representative of current server technology. Newer generation servers are trending towards better energy proportionality and idle power costs as low as 30% are expected in the near future. A lower idle power cost will only exaggerate the fraction of overhead due to interference and lead to even greater savings in PACMan.

Comparison with Optimal: To facilitate computation of the optimal, we use a small number, 16, of VMs, with equal proportion of VMs from each of the four

benchmarks. We vary the degradation constraint from 10% ($D = 1.1$), to as high as 50%[1]. Aside from the optimal, we also compare against a naïve method that does not quantitatively manage degradation but conservatively leaves every other core unused [20].

Figure 5 shows the energy overhead of the consolidation determined by PAC-Man, and by the naïve method, over and above the energy used by the optimal method. The proposed approach is within 10% of the optimal, and significantly better than the naïve approach currently in use.

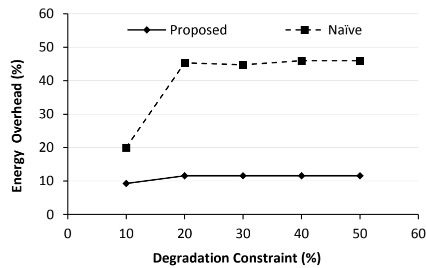

Figure 5: (P-Mode) Energy overhead comparison (computable for a small number of VMs). The overhead shown is the excess energy used compared to the optimal.

Figure 6 shows the server utilizations achieved by the three methods. The proposed method achieves over 80% utilization in most cases yielding good resource use. Of course, when the degradation allowed is small, servers must be left underutilized to avoid interference, and even the optimal method cannot use all cores.

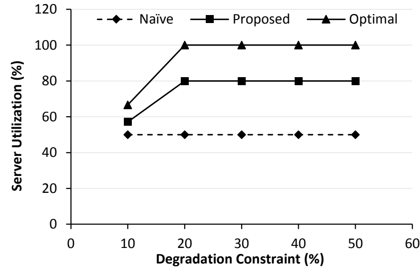

Figure 6: (P-Mode) Server utilizations achieved by the theoretical optimal, proposed, and naïve algorithms.

Large number of VMs: The second set of experiments uses more realistic

[1]If a degradation tolerance of 0% is enforced, then most VMs would require a dedicated machine, leading to an uninteresting (and inefficient) solution.

input sizes, up to $n = 1000$ VMs, again taking an equal proportion of VMs from each of the four SPEC CPU applications listed in Section 5.1. Since the optimal solution is not feasible to compute with a large numbers of VMs, we plot the resource overhead compared to the resources used when interference has no effect. In reality, interference will lead to a non-zero overhead and the optimal performance should be expected to be somewhere between 0% and the overhead seen for the proposed method.

Figure 7 shows the results, with a performance constraint of 50% ($D = 1.5$), for varying n. We see that the proposed method performs significantly better than the naïve one.

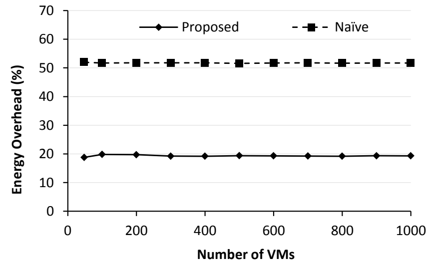

Figure 7: (P-Mode) Resource overhead comparison, normalized with respect to hypothetical resource use when there is no interference.

The utilizations achieved by the proposed method are between 75% and 80% for all n (plot omitted for brevity), as opposed to a server utilization of 50% achieved by the naïve method.

5.3 Eco-Mode

For the Eco-mode, we again compute the optimal solution for a small set $n = 16$ VMs with $m = 4$ servers, with the VMs taken from the SPEC CPU benchmarks. The initial allocation of VMs to servers is arbitrary and we repeat the experiment 10 times, starting with a random initial allocation each time. Since *any* allocation can be reached in at most $(k - 1)(m - 1) = 9$ swaps, we vary the number of allowed swaps G from 2 to 9. As an additional point of comparison we use a naïve approach that does not consider interference and places the VMs randomly. The performance of the randomized approach is averaged across 10 trials.

Figure 8 shows the excess degradation suffered by the VMs compared to that in the optimal allocation. The practical heuristic used in PACMan performs very close to the optimal and has up to 30% lower degradation than the naïve method.

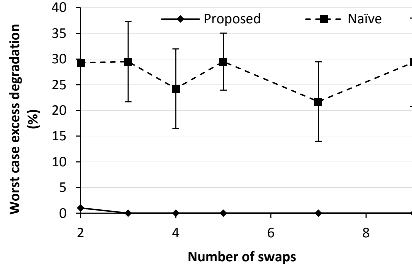

Figure 8: (Eco-mode) Maximizing worst case performance: Comparison of proposed heuristic and a naïve random algorithm with the theoretical optimal (computable for small input instances). Excess worst case degradation experienced compared to that seen in the optimal solution is shown. The error bars show the standard deviation across 10 random runs for the naïve approach.

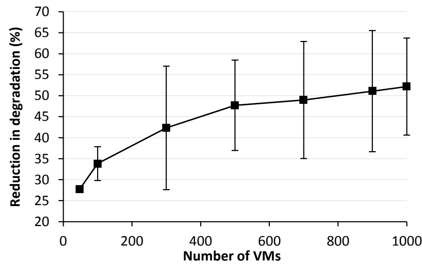

Figure 9: (Eco-mode) Large input size: Reduction in degradation compared to a naïve approach. The error bars show the standard deviation across 10 random placements for the naïve approach.

Next we vary the number of VMs up to $n = 1000$, packed tightly on $m = n/4$ quad core servers. The applications are taken from the SPEC CPU benchmarks as before, in equal proportion. The naïve approach used for comparison is a random placement that does not account for interference (10 random trials are performed for each point).

Since the optimal is not feasible to compute, we use the naïve approach as the base case and show the reduction in degradation achieved by PACMan (Figure 9). The worst case degradation is reduced by 27% to 52% over the range of number of VMs. While the number of servers is a fixed constraint, reduction in performance degradation results in a corresponding increase in throughput or reduction in runtime, yielding a proportional saving in energy per unit work performed.

In summary, we see that PACMan performs well on realistic degradation data.

5.4 TCO Analysis

The total cost of ownership (TCO) of a data center includes both the operating expenses such as energy bills paid based on usage, and capital expenses, paid upfront. Consolidation affects multiple components of TCO. The resultant savings in TCO are described below.

To compare capital costs and operating expenses using a common metric, James Hamilton provided an amortized cost calculation of an entire data center on a monthly basis [13]. In this calculation, the fixed costs are amortized over the life of the component purchased. For instance, building costs are amortized over 15 years while server costs are amortized over three years. This converts the capital costs into a monthly expense, similar to the operating expense.

Figure 10 shows the savings resulting in various data center cost components due to the proposed performance preserving consolidation method, compared to current practice [20]. In all, a 22% reduction in TCO is achieved, which for a 10MW data center implies that the monthly operating expense is reduced from USD 2.8 million to USD 2.2 million.

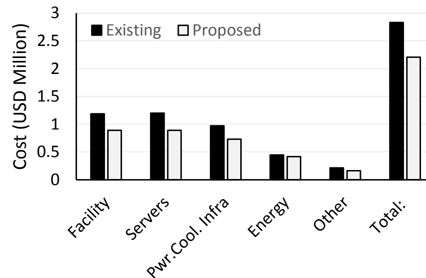

Figure 10: (P-Mode) TCO reduction using the proposed performance preserving consolidation method, compared to current practice [20] used in Google data centers. Pwr. Cool. Infra. refers to the power and cooling infrastructure cost, as defined in [13].

6 Related Work

We focused on performance degradation due to interference in the memory hierarchy. Performance isolation from memory subsystem interference has been studied at different levels of the system stack: the hardware level [26, 4, 32, 2, 15], the OS/software level [1, 22, 5, 25], and the VM scheduler level [3, 29]. These methods control the usage of shared resources to either improve isolation or tune excess

allocation at run time to compensate for degradation. Our method is complementary in that we facilitate determining the combinations of VMs that will interfere the least. The above techniques can then be applied to these preferred combinations, with lower resource overhead.

Performance estimates due to interference [18, 20, 11] have also been developed to aid VM placement. We build upon these works directly. The VM Profiling Engine in PACMan uses the interference characterization methods provided in these works.

Consolidation methods taking interference into account have been studied in [16, 27, 17]. These methods assume that servers are tightly packed and degradation is to be minimized, whereas we do not assume that servers are tightly packed. Rather, cores may be left unused to preserve performance. While only heuristics are provided in these works, we present a provably near-optimal algorithm. For the case where servers are tightly packed, we study an additional problem where the worst case performance is optimized. We show for this case that provably near-optimal methods are unlikely to exist.

7 Conclusions

We considered the problem of VM consolidation, one of the key mechanisms to improve efficiency for cloud infrastructures. In particular, we focused on the performance degradation that occurs due to resource contention among VMs, in spite of the isolation provided by current virtualization technologies.

The extent of interference and hence the resultant performance depends on both how many VMs are consolidated together on a server and which VMs are placed together. Hence, it becomes important to intelligently choose the best combinations. For many cases, performance is paramount and consolidation will be performed only to the extent that it does not degrade performance beyond the QoS guarantees required for the hosted applications. We presented a system that selects VMs that interfere the least with each other and places them together on the same server. This way the degradation is minimized, and as a result, the excess resource required to compensate for it is also minimized. While the problem of determining the best suited VM combinations is $\mathbf{NP}$-Complete, we proposed a polynomial time algorithm which yields a solution provably close to the optimal. In fact, the solution was shown to be within $\ln(k)$ of the optimal where k is the number of cores in each server, and is independent of the number of VMs, n. This is a very tight bound for practical purposes. We also considered the dual scenario where the resource efficiency is prioritized over performance. For instance, a certain number of servers is already provisioned and the objective is to get the best performance out of

them. For this case, we showed that even near-optimal algorithms with polynomial time complexity are unlikely to be found. Experimental evaluations showed that the proposed system performed well on realistic VM performance degradations, yielding over 30% savings in energy and up to 52% reduction in degradation.

We believe that the understanding of performance aware consolidation developed above will enable better workload consolidation in cloud platforms and virtualized servers. Additional open problems remain to be addressed in this space and further work is required to develop consolidation methods that operate in an online manner and place VMs near-optimally as and when they arrive for deployment in the cloud.

References

[1] Manu Awasthi, Kshitij Sudan, Rajeev Balasubramonian, and John Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of High-Performance Computer Architecture (HPCA)*, 2009.

[2] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the Symposium on Microarchitecture (MICRO)*, 2008.

[3] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proceedings of the International Symposium on Integrated Network Management*, 2007.

[4] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[5] Sangyeun Cho and Lei Jin. Managing distributed, shared L2 caches through os-level page allocation. In *Proceedings of the Symposium on Microarchitecture (MICRO)*, 2006.

[6] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[8] R. Duh and M. Fürer. Approximation of k-set cover by semi-local optimization. In *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, 1997.

[9] U. S. Environmental Protection Agency. Report to congress on server and data center energy efficiency public law 109-431. Technical report, EPA ENERGY STAR Program, 2007.

[10] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *SIGMETRICS*, pages 157–168, New York, NY, USA, 2009. ACM.

[11] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *ACM Symposium on Cloud Computing (SOCC)*, October 2011.

[12] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, dec 2008.

[13] James Hamilton. Cost of power in large-scale data centers. Blog entry dated 11/28/2008 at `http://perspectives.mvdirona.com`, 2009. Also in Keynote, at ACM SIGMETRICS 2009.

[14] R. Hassin and A. Levin. A better-than-greedy approximation algorithm for the minimum set cover problem. *SIAM Journal on Computing*, 2006.

[15] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. Vm3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, 2009.

[16] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, 2008.

[17] Yunlian Jiang, Kai Tian, and Xipeng Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *High Performance Embedded Architectures and Compilers*, pages 6–8, 2010.

[18] Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2007.

[19] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 959–967, 2007.

[20] 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 *(MICRO)*, 2011.

[21] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the USENIX international conference on Virtual execution environments (VEE)*, 2005.

[22] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 4th ACM European conference on Computer systems (EUROSYS)*, 2010.

[23] RightScale. RightScale scalable websites. `http://www.rightscale.com/solutions/cloud-computing-uses/scalable-website.php`.

[24] Eric Schurman and Jake Brutlag. Performance related changes and their user impact. In *O'REILLY: Web Performance and Operations Conference (Velocity)*, 2009.

[25] Livio Soares, David Tam, and Michael Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO*, 2008.

[26] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the conference on Power aware computing and systems*, HotPower, 2008.

[27] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the ACM International Conference on Computing Frontiers*, 2009.

[28] L. Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of the thirty-third annual ACM Symposium on Theory of Computing*, 2001.

[29] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the international conference on Supercomputing (ICS)*, 2008.

[30] VMware. Vmware distributed power management concepts and use. `http://www.vmware.com/files/pdf/DPM.pdf`.

[31] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, 2006.

[32] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *PACT*, 2007.