Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines

Sriram Govindan‡, Jie Liu†, Aman Kansal†, and, Anand Sivasubramaniam‡
‡CSE Dept., The Pennsylvania State University, †Microsoft Research, Redmond.
sgovinda@cse.psu.edu, {Jie.Liu,kansal}@microsoft.com, anand@cse.psu.edu

ABSTRACT

Workload consolidation is very attractive for cloud platforms due to several reasons including reduced infrastructure costs, lower energy consumption, and ease of management. Advances in virtualization hardware and software continue to improve resource isolation among consolidated workloads but a particular form of resource interference is yet to see a commercially widely adopted solution - *the interference due to shared processor caches*. Existing solutions for handling cache interference require new hardware features, extensive software changes, or reduce the achieved overall throughput. A crucial requirement for effective consolidation is to be able to predict the impact of cache interference among consolidated workloads. In this paper, we present a practical technique for predicting performance interference due to shared processor cache which works on current processor architectures and requires minimal software changes. While performance degradation can be empirically measured for a given placement of consolidated workloads, the number of possible placements grows exponentially with the number of workloads and actual measurement of degradation is thus not practical for every possible placement. Our technique predicts the degradation for any possible placement using only a linear number of measurements, and can be used to select the most efficient consolidation pattern, for required performance and resource constraints. An average prediction error of less than 4% is achieved across a wide variety of benchmark workloads, using Xen VMM on Intel Core 2 Duo and Nehalem quad-core processor platforms. We also illustrate the usefulness of our prediction technique in realizing better workload placement decisions for given performance and resource cost objectives.

1. INTRODUCTION

Multicore processors have become mainstream today–commodity processors already have 4-12 cores and are moving towards 10s-100s of cores [17]. The increased number of cores not only provides additional computational capacity to accelerate the performance of a single application, but also allows placing several applications within a processor. Cloud computing (CC) platforms [1, 41] exploit the latter capability to consolidate multiple applications within a server, resulting in reduced infrastructure costs and amortize server idle power costs. *Server virtualization* solutions such as *Xen*, *Hyper-V* and *Vmware* have also kept pace to accommodate high degrees of consolidation with support for heterogeneity of software stacks, fault isolation, and, dynamic resource scaling.

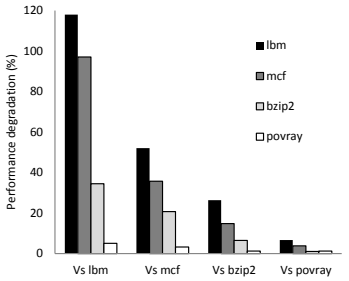


Figure 1: Performance degradation when a pair of SPEC-CPU applications are run on a Core-2-Duo processor. Each application is run on a dedicated core, with static partitioning of memory/disk space.

Successful consolidation in cloud platforms crucially depends on the ability to provide performance isolation among co-located applications. Current Virtual Machine Monitors (VMMs) or Hypervisors [39, 3] provide isolation guarantees on some of the server resources through strict CPU reservations and static partitioning of memory and disk space. Significant research has been dedicated towards providing isolation guarantees for other resources such as disk bandwidth [28, 44] and network bandwidth [15]. However, some resources are very hard to isolate. Examples include on-chip shared resources including cache space, interconnection network, and, memory bandwidth. System software has very little control over such resources and they are almost entirely managed by the hardware in a best effort fashion. As a result, co-located virtual machines (VMs) do suffer from interference, which leads to degraded performance compared to the performance achieved when running on a dedicated server, with the same resource reservation from the hypervisor. The extent of performance degradation depends on the application, context, and hardware making it hard to characterize [6, 38, 19, 46, 21, 5].

1.1 Performance Interference

In this paper, we focus on the performance impact of consolidated applications due to shared on-chip resources such as the last-level cache space and memory bandwidth. To illustrate the extent of performance degradation that could result from on-chip shared resource contention, we run a set of SPEC-CPU 2006 benchmarks co-located on a server with Intel Core-2-Duo processor (Figure 1).

We use Xen hypervisor and encapsulate each SPEC-CPU application within a linux virtual machine (VM). Each VM is assigned a dedicated core with its own statically partitioned memory and disk space. Performance degradation is measured by recording the increase in task finish time of the benchmark compared to its finish time when running alone with the same resource allocation (remaining core and memory left unused). With this definition, a 100% performance degradation means that the benchmark takes twice as long to finish when running consolidated compared to running alone. Since the applications have dedicated cores (eliminating private cache contention) and are not I/O intensive, it is reasonable to associate the reported performance degradation as a consequence of shared last-level cache and memory bandwidth. It can be seen from Figure 1 that on the Core-2-Duo processor, the performance degradation ranges from 2% (povray with povray) to 120% (lbm with lbm).

The extent of degradation clearly depends on the exact combination of applications that are co-located. For an effective consolidation policy, the level of interference among applications/VMs must be quantified. For instance, based on the above measurements, we can predict that given two dual core processors, the placement [lbm, povray] on one processor and [lbm, povray] on the other will give far better performance than [lbm, lbm] and [povray, povray], for the same resource use. Naïvely, performance interference due to consolidation can be computed by exhaustively measuring all possible VM placements. In this approach, a hosting platform consolidating M VMs with N VMs per server, needs to perform $M!/N!(M-N)!$ measurements. For a cloud computing platform provider, this can be prohibitively expensive.

Existing approaches to address on-chip performance interference largely falls into two categories: (i) *Architecture-level* solutions that propose hardware changes to expose cache usage information and provide isolation/QoS guarantees among threads [34, 37, 23, 46, 35, 27, 29, 43], (ii) *System-level* solutions that are primarily based on page-coloring techniques for enforcing cache partitioning across applications [7, 36, 45, 2]. While the architectural solutions are at the mercy of processor vendors to see applicability in practice, existing system-levels solutions are found to have very high implementation overheads (requiring changes at several layers of the software stack) [45] and cause inefficiency in the use of cache resources [42, 26, 9]. *More importantly, these cache partitioning approaches only control the application miss rate but not the miss latency.* Miss latency is due to contention for memory-bandwidth between the co-located applications (traffic between the last-level cache and DRAM memory banks) and is known to significantly impact overall application performance [22, 5, 20]. While analytical models [24] and simulation techniques can estimate the penalty due to memory bandwidth contention, applying such models across a variety of processor architectures and integrating their effects with application cache usage is non-trivial.

In this paper, we propose a novel and practical approach called *Cuanta* that uses active probing to precisely quantify the impact of shared cache interference and memory bandwidth contention for consolidated applications.

1.2 Proposed Approach

Problem Statement: Consider a cloud hosting service that consolidates M applications (VMs) on a set of servers with N-core processors, with at most one application per core[1]. The N applications that could potentially be co-located on a processor share the last-level cache (LLC) space and the memory bandwidth. Our goal is to estimate the performance degradation in each application due to shared chip-level resources (cache space and memory bandwidth) resulting from the remaining $N-1$ applications co-located on the processor.

Proposed Solution: Our approach, Cuanta, implements a unique way of estimating the cache usage behavior of applications through active probing and uses it to predict the performance degradation of applications upon consolidation with other applications. The key contribution of Cuanta is the design of a synthetic cache loader benchmark which can be tuned to generate fine-grained cache access patterns at the granularity of the available sets and ways of modern set-associative caches. Cuanta uses the synthetic cache loader benchmark to create a *cache clone* for each application entering the hosting platform. The purpose of an application's cache clone is to mimic its cache "pressure," which includes both the cache space and the memory bandwidth occupancy of the application. The cache clones are later used as a proxy for the actual applications when predicting performance degradation for different application co-location scenarios.

During the profiling phase, Cuanta generates the full interference vector for an application, i (that is, i's performance degradation when executed with *any* set of $N-1$ applications co-located on the other cores) by conducting a set of experiments co-locating i with just the cache clones. The number of such experiments required during this phase is only $(N-1) \times l$, where l is the number of unique clone applications[2].

A key aspect of Cuanta design is that the number of experiments required for profiling an application does not depend on the actual number of applications, M, hosted in the cloud, where M can be arbitrarily large. Other salient features of Cuanta include: (i) Cuanta works on existing processors and does not rely on the presence of cache-related hardware counters which are restricted in current hardware [40] (none of the existing processors expose cache usage information [46]) and therefore is easily portable across widely varying processor platforms; (ii) Cuanta does not require application instrumentation and unlike page-coloring based techniques does not require any changes to the software stack of the hosting platforms; (iii) Cuanta provides fine grained cache interference estimation at cache set and way granularity; and (iv) performance estimation does not require running the actual (potentially unbounded) applications among each other. The measurement overhead is linear in the number of cores sharing the LLC.

We evaluate Cuanta using the SPEC-CPU 2006 benchmark suite on Intel Core-2-Duo (2 cores) and Nehalem Quad-core (4 cores) processors, and predict application performance for a variety of co-location scenarios within 4% of the measured performance. The predicted performance degradation using Cuanta allows making intelligent consolidation decisions — such as *whether to consolidate* and *which VMs are better to co-locate for tolerable performance degradation*. We also present a use case that illustrates how the interference prediction results from Cuanta can guide VM consolidation. Using Cuanta, we predict the performance for various possible placements, and select the one that yields the appropriate energy efficiency and performance trade-off. Our technique is complementary to existing solutions for cache partitioning and can provide useful information about cache contention (at finer granularity of cache sets/ways) at runtime to OS paging and page coloring algorithms to provide better performance isolation guarantees.

[1]For simplicity, we do not consider time-sharing of applications within a core. Time-sharing may have additional interference such as the private-cache (L1) contention and scheduler related issues which we plan to address as part of our future work.

[2]l is a tunable parameter of our cache loader benchmark and can be varied to tradeoff prediction accuracy with the time required to characterize applications.

The remainder of the paper is organized as follows. We describe our cache usage characterization and mapping of applications to their corresponding cache clones in section 2. We develop prediction techniques for estimating the performance degradation of applications upon co-location in section 3. In section 4, we present mapping and prediction results for the SPEC-CPU 2006 benchmark suite. In section 5, we illustrate the applicability of this prediction for making intelligent consolidation decisions. Related work is discussed in section 6. Finally, we conclude and discuss directions for future work in section 7.

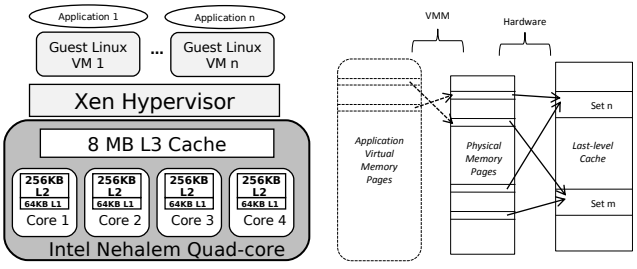
2. CHARACTERIZING INTERFERENCE

In this section, we give a brief overview of the processor cache hierarchy and describe Cuanta's methodology of characterizing the performance impact of consolidated applications due to chip-level shared resource interference.

2.1 Background on Cache Hierarchy

Today's multi-core processors have a hierarchy of caches, typically one or more caches private to each core and a single last-level cache (LLC) shared across all cores. As an example, Figure 2(a) shows a three-level cache hierarchy of the Intel Nehalem Quad-core processor. As shown in the figure, the processor has four cores, each with private 64 KB L1 and 256 KB L2 caches and a shared 8 MB last-level (L3) cache. Also shown in the figure is the software stack which includes the Xen [3] virtual machine monitor (VMM) or hypervisor hosting a set of applications, each contained within a guest virtual machine (VM). The guest VMs run their own operating system images and are provided guarantees on CPU cycles and memory size (statically allocated) by the VMM.

Figure 2(b) illustrates the mapping between applications' virtual memory to the processor cache for the software stack shown in Figure 2(a). Each application's virtual address space is mapped on to the corresponding guest VM's memory using the guest VM kernel's memory management software. The guest VM's memory is in turn mapped to the physical memory by the VMM. Finally, the physical pages are mapped to the processor cache structure by hardware. As we can see, several layers of software and hardware are involved in the mapping between applications' virtual memory to underlying cache structure, making it a difficult resources to manage.



(a) Hardware/Software Stack (b) Cache Mapping

Figure 2: (a) Intel Nehalem Quad-core processor with core-private L1 and L2 caches and a shared 8MB L3 cache. (b) Mapping of application's virtual memory pages to the cache structure via physical memory. Though the mapping between virtual memory and physical memory is managed by software, the mapping of data from physical memory to cache structure is entirely done by hardware.

Data is inserted and evicted from the cache hierarchy at the granularity of a *cache line* (64 bytes for most processors). The LLC is typically arranged in a set-associative fashion with certain number of cache sets and ways (8192 sets and 16 ways for the Intel Nehalem Quad-core processor in Figure 2(a)). Each memory address is direct-mapped to a set (typically data is mapped to a cache set using simple MOD arithmetic on the memory address) and then one of the ways within the set is associated with the memory address based on availability. Thus, placement of data in physical memory pages can be used to guide the eventual placement of data in the cache structure. This relationship between physical memory addresses and the cache structure is currently being used by system software techniques such as page coloring for partitioning cache space among applications. Cuanta exploits the above relationship to control fine-grain access to the cache structure and use it to infer application cache usage.

2.2 Synthetic Cache Loader (*scl*)

Cuanta uses a synthetic cache loader to profile an application's cache usage behavior and the extent of cache pressure it would exert on co-located applications. The synthetic loader is a tunable cache intensive workload that can access a specified region of the last-level cache (LLC) at the granularity of sets and ways. The loader, referred to as *scl*, accesses the memory pages allocated to itself in order to control the cache sets accessed by it. As explained in Section 2.1, controlling the memory addresses allows controlling the cache sets accessed. We use *scl(s,w)* to denote this synthetic cache loader workload tuned to access s sets and w ways in the LLC.

Implementation Issues: We implement our *scl* within a separate Linux VM. Implementing *scl* on a realistic guest kernel and VMM poses certain challenges because of the nature of fine grained memory access control involved, which is often abstracted out by the processor cache management and system memory management stacks. First, *scl* requires access to physical memory addresses (refer Section 2.1) to ensure access to specified number of cache sets and ways. In modern operating systems, user-level processes have access to only virtual memory addresses. Therefore we develop a kernel module inside the VM running the *scl* workload and use the Linux kernel function `'alloc_pages()'` for gaining access to physical memory pages from within the guest virtual machine. However, in a virtualized environment, a guest kernel's notion of physical memory is further abstracted from the *real physical memory pages* which are only controlled by the VMM. Therefore, we use a hypercall (`'HYPERVISOR_MEMORY_OP()'`) from inside *scl* VM kernel module to gain access to *real physical memory pages*.

Second, the cache space occupied by *scl* can be evicted by the cache manager. Hence, the *scl(s,w)* workload must continuously perform a read followed by a write on the specified s sets and w ways, keeping the cache hot and preventing these pages from getting evicted. Finally, the private cache hierarchy above the LLC also interferes with the cache accesses to LLC. Since the goal of *scl* is to create cache contention at only the shared cache, it avoids accessing data loaded or pre-fetched in the private caches by introducing an offset between consecutive accesses such that they miss on the private caches and access data only from the shared LLC.

Note that, all the functionalities of the *scl* workload are implemented within a separate VM and it does not require any modification to the VMs that actually host the applications (applications and their VMs are considered as a black-box in our approach).

2.3 Application Clones

In this section, we describe how the *scl* workload is used to infer the cache pressure exerted by an application. The procedure consists of tabulating an interference matrix for *scl* at different settings,

profiling the application against *scl* to obtain its interference vector, and then selecting the *scl* settings that clones the behavior of the application. Each of these three phases is described below.

2.3.1 Interference Matrix

The interference matrix records the performance degradation in *scl* at certain parameter settings when co-located with another instance of *scl*, at the same or other parameter settings. This is an off line process, and needs to be performed only once for each processor architecture. In a cloud setting this process would be performed once when servers with a new processor family are added to the hosting infrastructure. This process requires only a few minutes. The interference matrix is generated as described below.

Performance for *scl* is measured as number of bytes accessed per unit time (throughput metric). This can be measured every few milliseconds (over any duration long enough to cover a few repetitions of the cache access pattern used). Performance measurement for *scl* at each parameter setting thus requires a few milliseconds only and is not very time consuming.

To generate the *scl* interference matrix, we perform a set of measurements to record the performance degradation of a pair of *scl* (s,w) instances with different (s, w) settings. Suppose the set-associative LLC for the processor has m sets and n ways, then (s, w) can take any of $m \times n$ values. Since this can be very large, we use a subset of these values, periodically spaced in the entire range. For instance, for the Core-2-Duo, we use $s \in \{128, ..., k * 128, ..., 4096\}$ and $w \in \{3, ..., k * 3, ..., 24\}$ while for the Nehalem Quad-core, we use $s \in \{1024, ..., k * 1024, ..., 8192\}$ and $w \in \{1, ..., k * 1, ..., 16\}$. Let s_{max} and w_{max} denote the discrete number of sets and ways, respectively. Then $s_{max} \times w_{max}$ denote the number of unique settings for each *scl* instance. Applying these to both *scl* instances, a total of $(s_{max} \times w_{max})^2$ measurements are required, yielding a two dimensional matrix of performance degradations.

These discrete [sets, ways] pairs capture the entire spectrum of cache accesses with reasonable accuracy and are chosen based on insights gained from our empirical measurements. We tabulate

	Core 2 Duo (2 cores)	Nehalem Quad-core (4 cores)
Core-private caches	L1 (64 KB)	L1 (64 KB) L2 (256 KB)
Shared last-level cache	L2 (6 MB) 4096 sets, 24 ways	L3 (8 MB) 8192 sets, 16 ways
scl access granularity	128 sets, 3 ways	1024 sets, 1 way
Discrete combinations ($s_{max} \times w_{max}$)	256 discrete [sets, ways] pairs	128 discrete [sets, ways] pairs

Table 1: Salient details of our evaluation platform: Core-2-Duo and Nehalem Quad-core processors.

these details for the two hardware platforms evaluated in this paper, Intel Core-2-Duo and Intel Nehalem Quad-core processors, in Table 1. As shown, the Core-2-duo and Nehalem Quad-core processors need 256 and 128 unique experiments respectively. At 10ms per measurement, we need about 11 minutes to perform the pairwise measurements ($(s_{max} \times w_{max})^2$).

Interference vector: The above measurements are used to define an *interference vector* for each *scl* setting. Suppose on the processor of interest, the degradation in performance of *scl* (s_i, w_j) when running co-located with *scl* (s, w) is denoted $\Delta^*(i, j)$. Then the interference vector, which is a succinct representation of degradation caused by *scl* (s, w) in the performance of other *scl* instances, is defined as:

$$I_{(s,w)} = [\Delta^*(1,1), ..., \Delta^*(s_{max}, w_{max})] \quad (1)$$

2.3.2 Application Profile

Similar to measuring the degradation for *scl* when running with other instances of *scl*, we also measure the performance degradation of *scl* at various settings when running with the application. This profiling step needs to be performed once for each application and is independent of the number of other applications in the hosting platform. The performance degradation of *scl* is measured, which as mentioned before, requires only a few milliseconds and all settings can thus be covered in a few seconds.

After the above measurement is finished, we tabulate the obtained data for each application α, in the form of an interference vector, denoted I_α:

$$I_\alpha = [\Delta(1,1), ..., \Delta(s_{max}, w_{max})] \quad (2)$$

where $\Delta(i, j)$ denotes the performance degradation for *scl* (s_i, w_j) when co-located with application α.

2.3.3 Clone Selection

The interference vector for each application, I_α, acts as a cache pressure signature or degradation footprint and we use it to find an equivalent *scl* (s,w) workload that has a similar footprint. This essentially involves mapping the interference vector of an application to one of the interference vectors of *scl* in the interference matrix generated previously for the processor architecture. We used Euclidean distance to determine the closest match. Denoting the representative *scl* settings for application α with R_α, this may be stated as:

$$R_\alpha = \arg\min_{I_{(s,w)}} Dist(I_\alpha, I_{(s,w)}) \quad (3)$$

This R_α acts as the application clone.

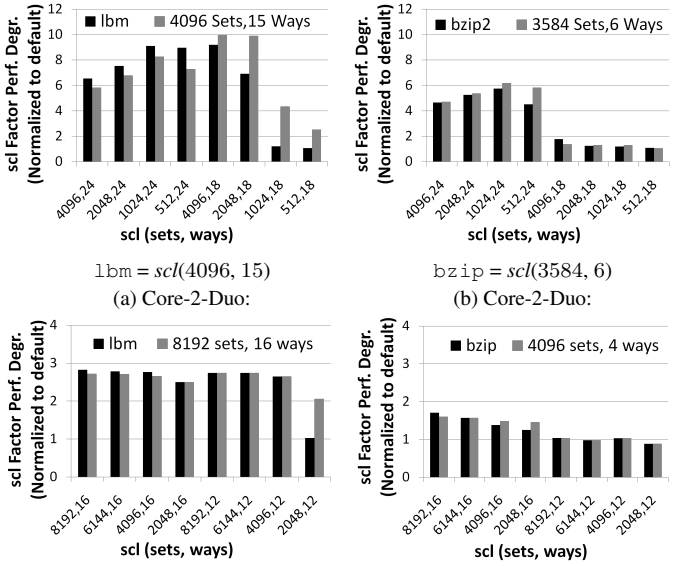
2.3.4 An Example

As an illustrative example, we plot the interference vector of two SPEC CPU applications, `lbm` and `bzip`, alongside the interference vector of the *scl* clones with the least Euclidean distance in Figure 3. Note that for clarity, only a few selected s and w values are shown for the representative vectors, instead of all elements (256 for Core-2-Duo and 128 for Nehalem Quad-core). The figure shows that the cache interference behavior of the applications is very similar to that of the mapped *scl* workloads.

2.4 Variations in Cache Pressure

Temporal Variations: The technique for inferring application cache pressure described above assumes that an application does not change its cache behavior over time. But in reality, applications do exhibit temporal variance in their behavior [18], with time scales varying from finer granularity of seconds or minutes to coarser granularity of hours or days. We capture such temporal variations by recording the interference vector shown in equation (2) not once but several times over the execution phases of the application (say, periodically every few seconds). We evaluate the efficacy of our temporal cache pressure inference in Section 4.

Spatial Variations: It is important to note that the exact cache sets (and not just the number of cache sets) allocated to co-located applications determines the degree of overlap in the cache sets and hence the degree of cache interference. Our synthetic cache loader *scl*(s,w) only indicates the number of cache sets occupied by the workload and there exists a combinatorial choice in selecting those s sets among the total number of cache sets. We measured the effect of selection of specific s sets by using a rolling window in our *scl*(s,w) workload and accessing all possible contiguous s sets. In most cases, the interference observed using the rolling window of



lbm = *scl*(4096, 15) (a) Core-2-Duo:

bzip = *scl*(3584, 6) (b) Core-2-Duo:

lbm = *scl*(8192, 16) (c) Nehalem Quad-core:

bzip = *scl*(4096, 4) (d) Nehalem Quad-core:

Figure 3: Interference vector mapping of SPEC CPU workloads, lbm (high cache pressure) and bzip2 (moderate cache pressure), on Core-2-Duo and Nehalem Quad-core processors. The interference vectors of lbm (I_{lbm}) and bzip (I_{bzip}) and the corresponding mapped *scl* workloads with the smallest Euclidean distance are shown. Performance degradation is normalized to performance when run alone.

cache sets showed low variance. Based on this observation, we account for spatial variation by taking the average interference across all rolling windows.

3. PREDICTING PERFORMANCE

In this section, we describe the use of interference characterization developed above for predicting performance degradation of consolidated applications. The procedure consists of characterizing the aggregate cache pressure of multiple co-located applications and using it to develop a performance degradation lookup table for each application when it enters the system.

3.1 Degradation Table

Consider a processor with n cores. Suppose n applications are to be consolidated on this processor. We wish to predict the performance degradations due to interference before actually performing the consolidation placement. To predict the performance degradation in any one of the n applications, we effectively need to model the effect of interference from the remaining $(n-1)$ applications.

Let α_i denote the i-th application, for $i \in \{0, ..., n-1\}$. First, note that using the cache clones developed earlier, the interference on α_0 from <α_1, α_2, ..., α_{n-1}> can be estimated by measuring the performance of α_0 when co-located with <R_{α_1}, R_{α_2}, ..., $R_{\alpha_{(n-1)}}$>, where R_{α_i} is the representative *scl* clone[3] for α_i defined in Equation 3. Note that application performance is being measured rather than *scl* performance and application performance can be obtained in hosting platforms using readily available performance counters such as Instructions Per Cycle (*IPC*).

If this measurement is performed for all *scl* settings (covering all possible R_{α_i} clones) and store the results as a lookup table, then we can predict the degradation for α_0 for any placement combination. This is a reduced number of measurements compared to measuring with each possible application since the number of *scl* settings is bounded while the number of applications in the hosting platform can grow unbounded. However, the number of measurements as described above is still exponential in n since all possible *scl* settings are to be covered for each of the $(n-1)$ *scl* instances, which is $(s_{max} \times w_{max})^{(n-1)}$ number of measurements.

3.1.1 Reduction of Measurements

We address the above issue by using a technique to map the aggregate cache pressure generated by multiple combinations of *scl* workload to a much smaller number of combinations, that results in reducing the number of measurements from exponential to linear in n. The key insight is that, the cache pressure exerted by any two *scl* workloads can in turn be mapped to that of a single *scl* workload (at higher parameter values) or to a combination of one *scl* at maximum parameter settings and another *scl* workload with variable settings. In effect, $s_{max} \times w_{max}$ number of variable settings are required to be measured for only one *scl* instance, instead of $(s_{max} \times w_{max})^2$ settings for the two *scl* instances. Suppose $I_{(s1,w1)+(s2,w2)}$ denote the interference vector for a pair of *scl* instances parametrized by (s_1, w_1) and (s_2, w_2), running simultaneously. The above insight can be stated as:

OBSERVATION 3.1. *The cache pressure generated by two* scl *workloads parametrized by* (s_{i1}, w_{j1}) *and* (s_{i2}, w_{j2}) *can always be mapped as either:*

$$\text{scl}(s_{i1}, w_{j1}) + \text{scl}(s_{i2}, w_{j2}) \rightarrow \text{scl}(s_{i3}, w_{j3}) \quad (4)$$

where,

$$(s_{i3}, w_{j3}) = \arg\min_{(s,w)} Dist(I_{(s_{i1},w_{j1})+(s_{i2},w_{j2})}, I_{(s,w)})$$

or:

$$\text{scl}(s_{i1}, w_{j1}) + \text{scl}(s_{i2}, w_{j2}) \rightarrow \text{scl}(s_{max}, w_{max}) + \text{scl}(s_{i4}, w_{j4}) \quad (5)$$

where,

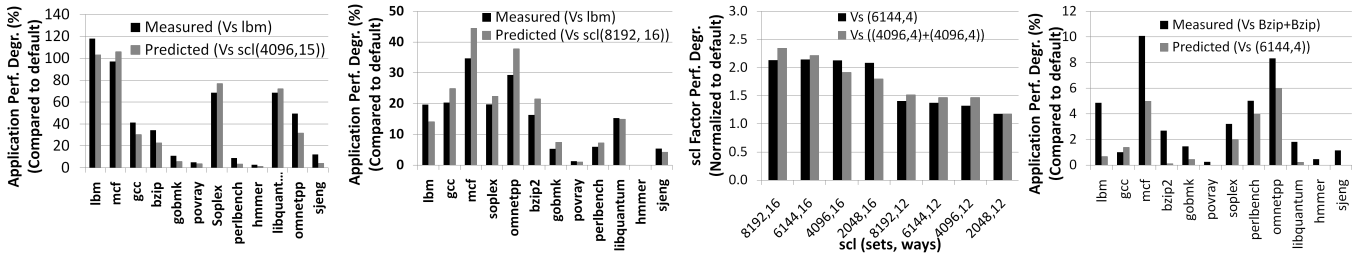
$$(s_{i4}, w_{j4}) = \arg\min_{(s,w)} Dist(I_{(s_{i1},w_{j1})+(s_{i2},w_{j2})}, I_{(s_{max},w_{max})+(s,w)})$$

In other words, for a 3 core processor, one of which hosts α_0 and the remaining two cores host 2 other applications, instead of measuring performance degradation with all *scl* settings for the remaining two cores, we only need to measure performance with (i) all possible settings of *scl* on one core (which effectively covers also the *scl* pairs satisfying the conditions for Equation (4)) and, (ii) *scl* (s_{max}, w_{max}) on one core and all settings of *scl* on the 3rd core (covering the combinations satisfying Equation (5)). The above reduction easily extends to more than 3 cores by recursively applying Equations 4 and 5.

Let $\delta_\alpha(scl_1(s_{i1}, w_{j1}), ..., scl_{n-1}(s_{i(n-1)}, w_{j(n-1)}))$ denote the degradation measured for application α when running with a combination of $(n-1)$ *scl* instances at settings (s_{ik}, w_{jk}) for $k \in \{1, ..., n-1\}$. Then, the degradation lookup table for application α, denoted D_α, is the set of entries represented by:

$$D_\alpha = [\delta_\alpha(scl_1(s_{i1}, w_{j1}), ..., scl_{n-1}(s_{i(n-1)}, w_{j(n-1)}))] \quad \forall s_i, w_j$$

[3]By actually running the *scl* clones as a proxy for the mimicked applications, our performance measurement not only capture the application cache miss rate, but also the latency due to memory-bandwidth contention from other clones/applications.



Vs `lbm` (a) Core-2-Duo: Vs `lbm` (b) Nehalem Quad-core: `2-bzip` Reduction Mapping (c) Nehalem Quad-core: Vs `2-bzip` (d) Nehalem Quad-core:

Figure 4: Predicted and measured performance degradations for workloads (along x-axis) when co-located with `lbm` on Core-2-Duo and Nehalem Quad-core processors, is shown in (a) and (b), respectively. Prediction is based on running the workloads (along x-axis) with the corresponding mapped *scl* clone workload for `lbm` in Figure 3. Figures (c) compares the interference vector of the original and reduced mapping when 2 instances of bzip application are co-located on the Quad-core processor. Figure (d) shows the measured and predicted performance degradations for workloads listed along the x-axis when co-located with the 2 instances of bzip application, where prediction is based on the reduced mapping (see Table 2).

which has $(s_{max} \times w_{max})^{(n-1)}$ entries corresponding to different *scl* settings. Using the above observation this reduces to:

$$D_\alpha = [\delta_\alpha(scl_1(s_{max}, w_{max}), ..., scl_m(s_{im}, w_{jm}))] \quad \forall s_i, w_j \quad \text{and} \quad \forall m \in \{1, ..., n-1\} \tag{6}$$

which requires only $(n-1)(s_{max} \times w_{max})$ measurements, based on $(s_{max} \times w_{max})$ settings of *scl* for each m.

Thus, instead of measuring performance degradation of an application for all possible co-location scenarios, the performance degradation for each application is measured for a bounded number of *scl* combinations. The number of measurements can be further reduced somewhat by using only those *scl* settings that comprise the already hosted applications on the cloud (rather than all *scl* settings). However, in that case, when a new application comes in and gets mapped to a new R_α, degradation tables for previously profiled applications also need to be updated with one more measurement for each.

3.1.2 Reduction Table

For the above reduction, we do need a mapping that relates the aggregate cache pressure of two *scl* instances to that corresponding to a single *scl* instance or the other *scl* instance required with *scl* (s_{max}, w_{max}). This mapping is obtained as follows: we run 3 copies of the *scl* workload (each run on a dedicated core) and use one *scl* workload to probe the cache pressure generated by the remaining two *scl* workloads. Let us denote the degradation in performance of *scl* (s_i, w_j) when running co-located with *scl* (s_{i1}, w_{j1}) and *scl* (s_{i2}, w_{j2}) as $\Delta_2(i, j)$. Then the interference vector, which represents the degradation caused by the aggregate cache pressure generated by *scl* (s_{i1}, w_{j1}) and *scl* (s_{i2}, w_{j2}) in the performance of other *scl* instances, becomes:

$$I_{(s_{i1},w_{j1})+(s_{i2},w_{j2})} = [\Delta_2(1,1), ..., \Delta_2(s_{max}, w_{max})] \tag{7}$$

For each $I_{(s_{i1},w_{j1})+(s_{i2},w_{j2})}$ we find the nearest $I_{(s,w)}$ from among all the interference vectors in the interference matrix of Section 2.3.1 (i.e., the setting (s_{i3}, w_{j3}) in equation (4)) and the $I_{(s_{max},w_{max})+(s,w)}$ vectors from the list of these recorded measurements themselves (i.e., the setting (s_{i4}, w_{j4}) in equation (5)).

The above measurements require a total of $(s_{max} \times w_{max})^3$ experiments for the three *scl* instances. At 10ms per measurement, we need about 46.6 hours (or 2 days) to construct the complete reduction table (assuming $(s_{max} \times w_{max})$=256). Though this may seem like a heavy overhead, this is a constant one time overhead which is done offline for each processor architecture and is *not* dependent on the number of processor cores. The above mapping is sufficient to map any number of *scl* instances with arbitrary settings to a set of *scl* (s_{max}, w_{max}) instances and just one *scl* instance with variable settings. For example, suppose we had a 6 core processor, to model the effect of 5 remaining VMs on α_0, we wanted to lookup the degradation entry at $< R_{\alpha_1}, R_{\alpha_2}, R_{\alpha_3}, R_{\alpha_4}, R_{\alpha_5} >$. Suppose the parameter settings for the clones R_i are the *scl* instances, $< scl_1, scl_2, scl_3, scl_4, scl_5 >$, using shorthand scl_i to denote *scl* (s_i, w_i). Then using the reduction table above, we can map scl_1, scl_2 to scl_{max}, scl_{12}, assuming the mapping was of the form of equation (5) without loss of generality. We can now use the same table to map scl_{12}, scl_3 to scl_{max}, scl_{123}. Proceeding in this fashion, the lookup required for $< scl_1, scl_2, scl_3, scl_4, scl_5 >$ becomes equivalent to a lookup required for $< scl_{max}, scl_{max}, scl_{max}, scl_{max}, scl_{12345} >$. This is already tabulated in the degradation table listed in equation (6).

3.2 Algorithm

The actual performance prediction for an application α placed with $(n-1)$ other applications, $\alpha_1, ..., \alpha_{(n-1)}$ boils down to looking up the entry from the degradation table corresponding to $< R_{\alpha_1}, ..., R_{\alpha_{(n-1)}} >$. The complete procedure for identifying the application clones and the degradation tables may be summarized as Algorithm 1, shown in Figure 5. The computational complexity is $O(1)$ for Step 1 and is independent of the number of cores or applications. Each of Steps 2 and 3 has $O(n)$ complexity where n is the number of cores, and is independent of the number of applications in the cloud. Step 4 has $O(n)$ complexity for each application's performance lookup. This is the key efficiency gain of Cuanta, requiring only $O(n)$ measurements per application, instead of exponential in n as would be the case for explicitly measuring the degradations for each placement.

3.2.1 Illustrations

As an illustration of the observation used for reducing the number of measurements required, Table 2 shows the mapping results for some combinations of the `lbm` and `bzip` SPEC CPU workloads in the quad-core processor.

To illustrate the use of degradation tables and reduction tables generated above for interference prediction, Figures 4(a) and (b) compares the predicted and measured performance degradation of several SPEC CPU applications when consolidated with `lbm` on

Algorithm 1. Predicting performance degradation using Cuanta.

1. **Initialize.** For the processor architecture in use, generate the:
   (a) *Interference Matrix:* Tabulate interference vectors $I_{(s,w)}$ for $s \in \{1, ..., s_{max}\}$ and $w \in \{1, ..., w_{max}\}$ (Section 2.3.1).
   (b) *Reduction Table:* Tabulate interference vectors $I_{(s_{i1},w_{j1})+(s_{i2},w_{j2})}$ for $s_{i1}, s_{i2} \in \{1, ..., s_{max}\}$ and $w_{j1}, w_{j2} \in \{1, ..., w_{max}\}$ and determine the reduction mappings (Section 3.1.2).
2. **Identify Clones.** For each workload α that enters the hosting infrastructure:
   (a) Record interference vector, I_α: Run workload and measure performance degradation of *scl* (s, w) for selected $s \in \{1, ..., s_{max}\}$ and $w \in \{1, ..., w_{max}\}$ (capture temporal variations if any).
   (b) Determine R_α: Find the interference vector(s) $I_{(s,w)}$ that is (are) closest to I_α. For workloads mapping to different interference vectors during different temporal phases, multiple $I_{(s,w)}$ may comprise R_α.
3. **Characterize.** For each workload α record its degradation table, D_α, defined in Equation (6), by measuring performance degradation of α with required *scl* (s, w) combinations.
4. **Predict.** For every combination $[\alpha_0, \alpha_1, ..., \alpha_{(n-1)}]$ that is determined to be a viable candidate placement by the consolidation system, lookup the degradation tables of $\alpha_i \forall i \in \{0, ...n-1\}$ at equivalent clone *scl* settings found from the reduction table.

Figure 5: Algorithm for performance prediction.

Core-2-Duo and the Nehalem Quad-core processors, respectively. The predicted results are obtained by measuring the performance degradation of the SPEC CPU workloads co-located with the equivalent cache clones for `lbm` identified in Figure 3. Only two cores of the quad-core processor are used for this experiment. Figures 4(c) compares the interference vectors of the original and reduced mapping for the aggregate cache pressure generated by 2 instances of the bzip workload. Figures 4(d) uses the reduced mapping to predict performance degradation of the 12 SPEC-CPU workloads when co-located with 2 instances of the bzip application. Though not shown here, we find the interference vectors of all the *scl* workload combinations between the 'Original Mapping' and 'Reduced Mapping' closely match against each other.

4. EVALUATION

We evaluate Cuanta cache pressure characterization and performance prediction for 12 SPEC-CPU 2006 workloads with diverse cache characteristics on the Intel Core-2-Duo and Nehalem Quad-core processors. We then discuss and evaluate the efficacy of our techniques in capturing temporal variation in application cache usage. Each application is run in a Linux 2.6.18 guest kernel hosted in a VM, using the Xen VMM 3.3.4 hypervisor, and is allocated a dedicated core. The SPEC-CPU 2006 benchmarks are CPU intensive with little I/O, and hence interference mainly arises in the LLC and memory hierarchy as opposed to disk or network bandwidth. Throughout this section, the reported prediction error corresponds to the absolute difference between the measured and estimated application performance degradation.

Applications	Original Mapping	Reduced Mapping
bzip+bzip	(4096,4)+(4096,4)	(6144,4)
bzip+bzip+bzip	(4096,4)+(4096,4)+(4096,4)	(6144,16)
lbm+bzip+bzip	(8192,16)+(4096,4)+(4096,4)	(8192,16)+(6144,4)

Table 2: Reductions using equations (4) and (5) for `lbm` and `bzip` in the Nehalem Quad-core processor. For this processor, (8192,16) corresponds to *scl* (s_{max}, w_{max}).

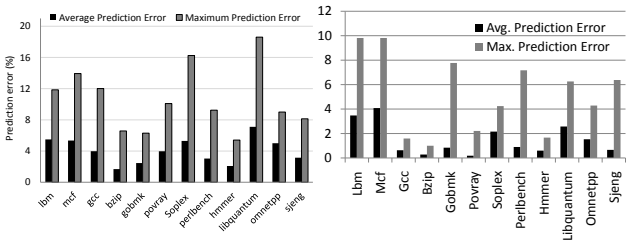
Table 3 presents the representative cache clones (as derived using the methods discussed in Section 2) for all the 12 applications on both the hardware platforms. It is worth noting that the cache clone mapping (*scl*(sets,ways)) of workloads might not reflect their real cache usage. Our intention is not to infer the exact cache usage but only to mimic the workload cache pressure. Therefore, our mapping of workloads to their respective cache clones occupying a specific number of cache sets and ways should not be interpreted literally. Without hardware support, there is currently no way to confirm that an application uses a specified number of cache sets and ways. The key metric that we use to evaluate the efficacy of the cache clones is in their ability to predict application performance upon consolidation, which we present later in this section. It is also important to note that the cache pressure exerted by an application is very different between the two hardware platforms, because of their different private cache sizes (see Table 1). Notably, while `bzip`, `gobmk`, and `sjeng` have similar cache pressure on the Core-2-Duo architecture, their characteristics are very different on Nehalem —`sjeng` is able to contain most of its memory access within the 256 KB L2 cache and has a negligible (zero) cache pressure at the LLC.

Application	Core-2-duo cache clone *scl* (sets, ways)	Nehalem Quad-core cache clone *scl* (sets, ways)
lbm	(4096, 15)	(8192, 16)
mcf	(2048, 24)	(6144, 16)
soplex	(3072, 15)	(8192, 12)
gcc	(3840, 6)	(4096, 4)
perlbench	(3840, 6)	(2048, 4)
hmmer	(3840, 6)	(4096, 4)
povray	(2048, 6)	(0, 0)
bzip	(3584, 6)	(4096, 4)
gobmk	(3584, 6)	(2048, 4)
sjeng	(3584, 6)	(0, 0)
libquantum	(3584, 15)	(8192, 12)
omnetpp	(1792, 24)	(8192, 6)

Table 3: Cache clones for 12 of the SPEC CPU 2006 suite applications in Core-2-duo and Nehalem Quad-core processors.

First, we present prediction results for 2-application consolidation scenario in Figure 6 when the 12 SPEC-CPU applications are co-located among each other in both Core-2-Duo and Nehalem processors. Each workload shown along the x-axis is run with all the 12 workloads (including itself) leading to a total of 12×12 measurements. The resulting average and maximum prediction error is reported for each of the workloads. The average prediction error was less than 7% and 4% across these workloads for the Core-2-Duo and Nehalem architectures, respectively.

Next, we investigate the prediction accuracy for 3-application and 4-application consolidation scenarios in the Nehalem Quad-



(a) Core-2-Duo prediction (b) Nehalem Quad-core prediction

Figure 6: Two-core consolidation: Average and maximum error in performance prediction for 12 SPEC-CPU workloads when consolidated with every other workload (including itself) on Core-2-Duo and Nehalem Quad-core processors.

core processor. The total number of possible placements is of course very large for 3 and 4 co-located applications (greater than 20,000 combinations). Rather than measure all possible combinations, we randomly selected 200 possible placements for each of the 3 and 4 core consolidation scenarios.

For space reasons we do not list all 200 combinations but a few select ones in Table 4. We present the average and maximum prediction error over all 12 workloads running on the last core, given that the first 2 or 3 cores run the workloads shown in the table.

Figure 7(d) shows the results across all 200 measurements for each of 3 and 4 core experiments in the form of a cumulative distribution function (CDF). The prediction results for the 144, 2-core consolidation experiments is also included (7(b)). The performance degradation CDF for both dual-core and quad-core processors (Figures 7(a) and 7(c) respectively) show that cache-interference can lead to undesirable performance degradation for a significant fraction of workload co-location scenarios. We see that almost a quarter of workload co-location scenarios can result in performance degradation of more than 25% for both the processors (except for the case where only 2 of the 4 cores in the quad core processor are used, where degradation is less than 25% for 95% of the co-locations).

Two observations can be made from the data in Figure 7. First, the error plots in Figures 7(b) and (d) show that Cuanta achieves a useful prediction accuracy: the error is less than 8% in almost 90% of the workload combinations, for both processor architectures.

Second, as seen from Figures 7(a) and (c), there exists a wide variance in the amount of degradation across the consolidation scenarios. While the maximum performance degradation exceeds 100% for both the processors, there exist a significant fraction of co-locations with less than 10% performance impact. This suggests that the performance prediction capability provided by Cuanta, if used for intelligent placement, could be very effective in improving the overall performance for a given number of processor cores in the hosting infrastructure.

4.1 Variation in Cache Usage

Recall from Section 2 that we address the temporal variation in application cache usage by performing the cache pressure probing several times during the application execution. Let $I_\alpha(i)$ denote the interference vector of workload α at the i^{th} interval of its execution duration. Based on the Euclidean distance for interference vector at each interval i, we estimate the fraction of time an application maps to a particular clone. Among the 12 SPEC-CPU applications, we find that `mcf` exhibits a significant temporal variation on the Core-2-Duo processor: `mcf` maps to $scl(2048, 18)$ for 30% of the time and maps to $scl(3072, 12)$ for 70% of the time when probed periodically every 30 seconds during its execution. We represent the resulting cache clone mapping as, R_{mcf} = { $(0.3, scl(2048, 18))$, $(0.7, scl(3072, 12))$ }.

Figure 8 shows the predicted and measured degradations for `mcf` in the Core-2-Duo processor. The result for prediction without temporal variance corresponds to mapping `mcf` to a single interference vector ((2048, 24) as shown in Table 3) based on average interference across its entire run. The figure shows the significance of temporal variance analysis in performance prediction accuracy. We do not find significant temporal variation for the other 11 applications in the Core-2-Duo architecture and none of the applications exhibited such variation in the Nehalem architecture.

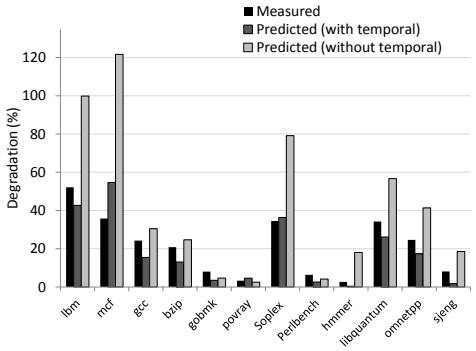


Figure 8: Predicted and measured degradations for workloads listed along the x-axis when co-located with `mcf`, an application that exhibits temporal variation in cache usage pattern. The error in prediction is much higher when the temporal variance is not accounted for.

SPECjbb load-level	Transactions per second	Cache clone (sets, ways)
20%	3967	(2048, 2)
50%	9918	(2048, 6)
100%	19836	(8192, 4)

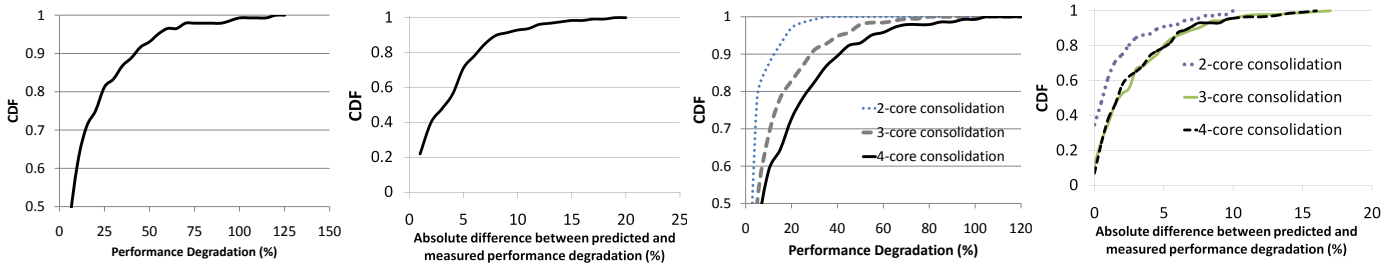
Table 5: Varying workload intensity of SPECjbb translates in to widely different cache interference.

Another source of variation in cache usage is change in workload volume, such as due to changes in the number of incoming user requests over the course of a day. We study such variation with a request driven enterprise workload, the SPECjbb 2005 benchmark. The amount of work performed by a SPECjbb VM changes with the number of transactions processed, which we vary to simulate temporal load variations. It is important to note that, unlike `mcf` where variations can be observed at the granularity of a few seconds primarily due to the change in nature of computation in various phases of the workload, the workload volumes that cause the behavior of enterprise applications to change would change only gradually, such as the granularity of an hour. Table 5 presents the mapping of SPECjbb to different cache clones based on its workload intensity, represented as the number of transactions processed per second.

We present performance degradation of the 12 SPEC-CPU workloads when co-located with 3 instances of the SPECjbb application at two different load-levels (20% and 100%) on the Nehalem quad-core processor in Figure 9. It can be seen that the performance degradation of the SPEC-CPU applications varies by a factor of 3 between the two SPECjbb load-levels, indicating that it is important to identify and react to these different workload phases. As part of our future work, we plan to investigate techniques to dynamically

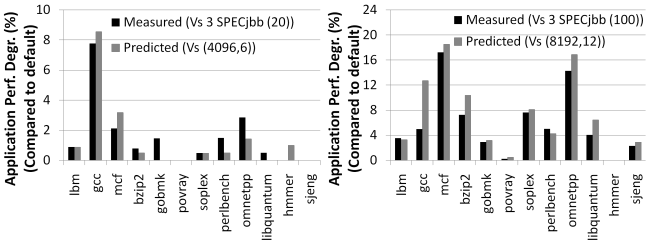
Co-located Applications 3-core	Cache clones	Predict Error (%) Max. [Avg.]		Co-located Applications 4-core	Cache clones	Predict Error (%) Max. [Avg.]
lbm+lbm	(8192,16)+(8192,16)	8.4 [4.0]		lbm+lbm+lbm	(8192,16)+(8192,16)+(8192,16)	7.1 [2.9]
bzip+bzip	(6144,4)	8.3 [3.0]		bzip+bzip+bzip	(6144,16)	9.5 [4.0]
gcc+soplex	(8192,12)	5.9 [2.5]		omnetpp+omnetpp+omnetpp	(8192,15)	6.8 [2.5]
mcf+mcf	(8192,12)	6.7 [3.1]		soplex+soplex+soplex	(8192,16)+(8192,16)+(8192,8)	15.7 [4.6]
mcf+bzip	(6144,16)	8.4 [2.5]		gcc+gcc+gcc	(6144,16)	7.0 [2.6]
gobmk+gobmk	(4096,4)	2.9 [0.9]		omnetpp+soplex+soplex	(8192,16)+(8192,12)	13.7 [3.1]
omnetpp+soplex	(8192,15)	6.2 [1.6]		omnetpp+omnetpp+lbm	(8192,16)+(8192,12)	5.6 [1.9]
omnetpp+gcc	(6144,16)	9.2 [3.5]		gcc+soplex+lbm	(8192,16)+(8192,12)	7.3 [2.5]
lbm+gcc	(8192,16)+(4096,4)	16.1 [4.5]		soplex+soplex+gcc	(8192,16)+(8192,10)	7.1 [1.9]
omnetpp+lbm	(8192,16)+(8192,6)	15.6 [7.0]		povray+povray+povray	(0,0)	2.2 [0.5]
soplex+soplex	(8192,16)+(8192,10)	4.9 [1.8]		hmmer+hmmer+hmmer	(6144,16)	15.0 [5.6]
lbm+soplex	(8192,16)+(8192,12)	14.3 [5.8]		gobmk+gobmk+gobmk	(6144,4)	6.8 [1.9]

Table 4: Mapping and prediction results for 3-application and 4-application consolidation scenarios in the quad-core processors. The 12 SPEC-CPU applications are co-located with application combinations specified in the first column. The difference between the measured and predicted performance degradation for the 12 experiments corresponding to each application combination (both average and maximum errors) are presented.



Performance Degradation (a) Core-2-Duo | Prediction Error (b) Core-2-Duo: | Performance Degradation (c) Nehalem Quad-core: | Prediction Error (d) Nehalem Quad-core:

Figure 7: CDF of performance degradation and prediction error for Core-2-Duo (a,b) and Nehalem Quad-core (c,d) processors. We present results for all possible combinations of the 12 SPEC-CPU applications for 2-core consolidation scenario in both Core-2-Duo and Nehalem Quad-core processor (144 such combinations). For 3-core and 4-core consolidation results, we randomly choose 200 workload combinations in each. We were able to predict performance for 90% of the workload co-location scenarios with less than 8% error.



(a) Vs 3 SPECjbb at 20% load each (b) Vs 3 SPECjbb at 100% load each

Figure 9: Measured and predicted performance degradation of the 12 SPEC-CPU workloads when co-located with 3 SPECjbb workloads with 20% load-level (a) and 100% load-level (b) on the Nehalem Quad-core processor.

identify these workload phases and re-configure workload placement by leveraging our prediction framework.

5. USE CASES

Cuanta can answer several key questions that may arise when improving the consolidation algorithms for cloud platforms, such as (i) What should be the right degree of consolidation to arrive at the best resource-performance trade-off? (ii) What applications can or cannot be co-located for satisfying performance SLAs? (iii) Among multiple placement choices that satisfy the consolidation constraints of resource reservation, efficient packing, and distribution across redundant servers, which choices would provide the best performance? (iv) Can the cache interference information be exposed to system software (such as for page-coloring) for making intelligent resource partitioning decisions?

Though an in-depth design of consolidation algorithms and placement strategies is beyond the scope of this paper, we discuss two simple use-cases to illustrate the applicability of Cuanta in assisting VM consolidation.

5.1 Energy-Performance Tradeoff

Workload consolidation reduces overall energy consumption by co-locating applications on as few active servers as possible and switching off the rest. This saves the idle power cost of under-utilized servers. However, it also results in undesirable performance degradation for certain applications. It is important for a hosting platform to predict the performance impact of any arbitrary co-location to arrive at the right energy-performance trade-off.

We illustrate this trade-off using eight benchmarks from the SPEC-CPU 2006 suite with moderate cache interference mcf, gcc, $bzip$, $gobmk$, lbm, $soplex$, $libquantum$, $omnetpp$. One instance of each application is to be placed on a set of Core-2-Duo based servers. We vary the number of servers allocated from 4 (maximum packing) to 8 (each instance gets a dedicated server). Figure 10 reports the performance and energy use assuming the best possible placement choice is made in terms of interference maximization for each placement using Cuanta. Performance and energy numbers shown are normalized with respect to these metrics for applications run-

ning on a dedicated server each. The idle and peak power consumption of our server is 50W and 90W respectively, and the figure assumes a linear relationship between CPU utilization and power within this range [14, 13].

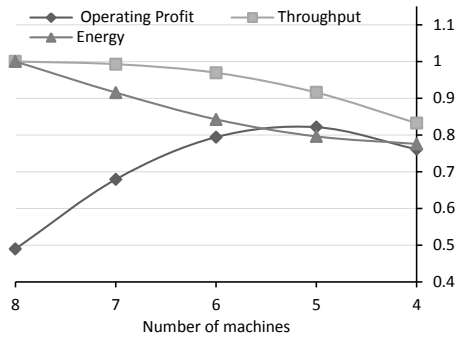


Figure 10: Resource and performance trade-off for interference aware consolidation. Performance prediction using Cuanta is used for estimating the degradation in overall throughput.

For a given performance SLA constraint, the degree of consolidation can directly be read off from the above curve. For instance, if required SLA is within 90% of peak performance, then using 5 machines will minimize energy use, while only 4 machines should be used if required SLA is only 80%.

Alternatively, a joint resource and performance utility metric may be optimized. Suppose the cost of energy is 10.5c per kWh (US average energy price [12]) and the revenue is 12c/hr when providing non-degraded throughput (similar to cost of one VM instance in commercial clouds). The operating profit, that is the difference of revenue and energy cost, is also shown in Figure 10. In this scenario, we see that consolidation helps up to 5 machines, but consolidating further is not advantageous. The best performance for 5 machines was achieved for the placement: $mcf - omnetpp$, $gcc{-}bzip$, $soplex{-}gobmk$ run in pairs using up 3 machines, while lbm and $libquantum$ run on a dedicated machine each. Thus, the technique to predict performance degradation can be used to determine when, and to what extent, consolidation is advantageous.

5.2 SLA Adherence

One of the main concerns for customers hosting their applications on the cloud is the fluctuation in performance over time, as has been measured for commercial clouds [31, 11]. In spite of resource reservations, certain events change the shared resource availability, such as (a) re-configuration of workload placement, triggered by addition of new applications to the cloud, (b) change in workload volume of co-located applications with time-of-day, etc. These variations affect the interference among co-located workloads leading to performance fluctuations.

We illustrate the applicability of Cuanta in making intelligent placement decisions while respecting application SLAs. Consider the following applications, $SPECjbb_1$, $SPECjbb_2$, $SPECjbb_3$, lbm_1, lbm_2, lbm_3, lbm_4, and lbm_5, to be placed in a hosting platform. The applications are given a number subscript to differentiate between the instances. We treat $SPECjbb$ instances that are transaction processing applications to be customer facing services that have a SLA constraint of 80% of their normalized throughput. Suppose that they initially start with an input request volume of 50%. The five lbm instances represent batch processing tasks and are run in best-effort mode (without SLA constraints). The cloud platform attempts to run the applications on as few servers as possible while satisfying SLA constraints. We assume that all servers use Nehalem Quad-core processors. We tabulate the different actions taken by hosting platform as the applications enter the system in Table 6. We associate a timestamp, t_i, with each event to indicate ordering (t_{i+1} happens after t_i). The *scl* interference vectors (as indicated in Equations (1) and (7)) are assumed to have been initialized for the Nehalem processor.

Event	Action/Observation
t_1: lbm_1, lbm_2, lbm_3, and $SPECjbb_1$ enter the cloud platform	(i) Measure interference vector I_α, find clone R_α, and construct degradation table D_α for each accepted application α. (ii) Place lbm_1, lbm_2, lbm_3, and $SPECjbb_1$ on server S_1. (iii) Perf. degr. of $SPECjbb_1$ predicted using Cuanta is 6% (within SLA).
t_2: lbm_4, $SPECjbb_2$, $SPECjbb_3$, and lbm_5 enter the cloud platform	(i) Measure interference vector I_α, find clone R_α, and construct degradation table D_α for each newly accepted application α. (ii) Place lbm_4, lbm_5, $SPECjbb_2$, and, $SPECjbb_3$ on server S_2. (iii) Perf. degr. of $SPECjbb_2$ and $SPECjbb_3$ predicted using Cuanta is 4% (within SLA).
t_3: Load intensity of $SPECjbb_1$ changes from 50% to 100%	(i) Perf. degr. of $SPECjbb_1$ measured as 40% (violates SLA). (ii) Measure interference vector $I_{SPECjbb_1}$, find clone $R_{SPECjbb_1}$, and construct degradation table $D_{SPECjbb_1}$ (iii) Migrate $SPECjbb_1$ to server S_2 and migrate $SPECjbb_2$ to server S_1. Xen live migration took about 2 minutes. (iv) Perf. degr. of $SPECjbb_1$ and $SPECjbb_2$ predicted using Cuanta is 15% and 6% (within SLA).

Table 6: Events and the corresponding actions taken by hosting platform by leveraging of Cuanta.

Change in workload intensity of $SPECjbb_1$ at time t_3 triggers re-characterization of the workload since we may not know the cache pressure exhibited by the new workload phase. Thus Cuanta can be used in dynamic decision making and for precisely identifying the target server to co-locate a new workload.

6. RELATED WORK

Cache is an essential part of modern processor for improving software execution performance. However, being an opportunistic mechanism, cache brings complexity to provide predictable performance and quality-of-service guarantee. Based on the underlying platform, we classify existing work into three categories.

Architecture-level solutions. Changes to processor cache architecture and intelligent cache replacement algorithms have been proposed to provide cache partitioning and QoS guarantees for co-located applications [25, 37, 23, 46, 9, 8, 34, 35, 27, 29, 43]. Since these approaches require alteration to the hardware design, they are typically evaluated in simulation. Qureshi et al. [28] present a cache utility function that captures an application's miss-rate as a function of cache-space allocation and propose a greedy technique to deal with non-convex behavior. Bitirgen et al. [5] use machine learning technique to understand the complex interaction between cache partitioning, memory bandwidth and CPU power states. The hardware-based solutions add complexity to processor design and the interface is hard to maintain stable over time. In comparison, software-based solutions have less processor dependencies.

OS/Software solutions. Software solutions to cache management mainly involves some variations of page coloring/re-mapping techniques for providing performance isolation [45, 2, 26, 7, 10, 4, 32, 30, 36, 33, 45]. Intelligent processor scheduling (mapping of threads to processor cores) have also been proposed to reduced

cache contention among co-located threads [47]. Lin et al. [22] observe that application performance is more sensitive to main memory latencies than shared-cache space and propose a dynamic cache partitioning technique that makes use of this observation to adjust application cache space based on their QoS requirements.

Measurement/Modeling based solutions. The techniques that are closest to our approach are measurement or modeling based solutions. These solutions treat the hardware and OS as black (or gray) boxes, and use external measurements to derive user observable performance models. Since they are not processor or OS dependent, these solutions are in general less accurate but are more widely applicable than the previous two kinds of solutions. In [21], Koh et al. propose a technique for predicting performance degradation of co-located applications based on their resource usage statistics, similar to program similarity analysis [16]. They use a resource usage vector that includes statistics such as cache hits/misses, processor utilization, and disk/network usage to describe an application. When a new application is hosted, its resource vector is compared with that of known applications (applications which are already profiled) and is mapped to the weighted average of one or more known applications whose resource vectors it closely resembles. Then the performance degradation of the new application is predicted based on already recorded performance degradation of the mapped/representative known applications. West et al. [40] looked at using hardware performance counters for estimating cache occupancy and use analytical models to predict increase in cache miss upon co-placement. In [38], Verma et al. assume that cache occupancy is explicitly provided by the HPC applications and provide heuristics for co-locating applications with minimal impact on cache interference. Another approach to managing performance interference by measuring it in situ was presented in Nathuji et al. [26]. That approach uses an online MIMO model to capture the performance interference effects in terms of resource allocations. The method then adjusts the processor allocation for each application based on the required performance level, in effect compensating for performance degradation by allocating additional resources.

7. CONCLUSION AND FUTURE WORK

The intention of this paper is to develop a software only solution to characterize cache interference of consolidated applications. Cache usage information, which is not exposed in current processor architectures, is very crucial in determining the performance impact of consolidated applications. Although performance degradation of consolidated applications can be empirically measured for making placement decisions, the number of possible workload placements is combinatorial in the number of workloads, rendering it impractical. We developed a novel technique to estimate the cache usage of co-located applications and use it to predict the performance degradation of applications due to cache interference. Our technique work on current processors without requiring changes to the software stack of the hosting platform and the prediction overhead is only linear to the number of cores sharing the last-level cache. Our evaluation results on Intel Core-2-Duo and Nehalem Quad-core processors are very positive and we were able to predict the performance degradation of 12 SPEC-CPU 2006 applications with very high accuracy of 96%.

There are several interesting directions for future work, (a) The cache pressure exerted by an application is dependent on several other factors such as its CPU reservation, CPU frequency/voltage level, private-cache interference etc., and we plan to extend our probe-based characterization to incorporate these factors. (b) Apart from on-chip resource interference, several other forms of interference such as storage device (hard-drive/SSD) interference, network bandwidth contention, etc., also contribute to the overall application performance. We wish to address resource interference across these different layers and capture their inter-dependency to develop a holistic framework for predicting performance degradation due to consolidation. (c) Our fine-grained probing of application cache usage can provide useful runtime information about the degree of cache contention to system-software such as page-coloring/remapping techniques, which can in turn use this information for providing better performance isolation guarantees. (d) We would also like to evaluate the impact of cache pressure between applications that do not share the last level cache space. This could happen due to just the memory-bandwidth contention between applications co-located on the same server but on different processor sockets. Our *scl* based characterization can be readily extended to incorporate such scenarios. (e) Characterization of temporal variation in application cache usage, though addressed in this paper, needs further attention. Especially, the granularity of variance impacts the profiling duration/accuracy. Techniques for detecting dynamic phase change of applications and consequent reconfiguration decisions are part of future work. (f) Evaluating the efficacy of our prediction technique for enterprise (multi-tier, possibly spanning multiple physical servers) workloads such as TPC-W, RUBIS, streaming-media servers, etc., are also of interest.

8. REFERENCES

[1] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[2] M. Awasthi, K. Sudan, R. Balasubramonian, and J. 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.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebuer, I. Pratt, and A. Warfield. Xen and the Art of Virtulization. In *Proceedings of the Nineteenth Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] B. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[5] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[6] N. Bobroff, A. Kochut, and k. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Proceedings of the International Symposium on Integrated Network Management*, 2007.

[7] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[8] D. Chandra, F. Guo, S. Kim, and Y. 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.

[9] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the international conference on Supercomputing (ICS)*, 2007.

[10] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[11] J. Dejun, G. Pierre, and C.-H. Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In

Proceedings of the international conference on Service-oriented computing, 2009.

[12] Department of Energy - prices and trends. http://www.energy.gov/pricestrends/index.htm.

[13] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system power analysis and modeling for server environments. In *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[14] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2007.

[15] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the Conference on Middleware (MIDDLEWARE)*, 2006.

[16] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the conference on Parallel architectures and compilation techniques (PACT)*, 2006.

[17] Intel Corporation. Intel Single-chip Cloud Computer. `http://techresearch.intel.com/spaw2/uploads/files/SCC-Overview.pdf`.

[18] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the Symposium on Microarchitecture (MICRO)*, 2006.

[19] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. Vm3: Measuring, modeling and managing vm shared resources. *Computer Networks*, 53(17):2873–2887, 2009.

[20] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *Proceedings of High-performance Computer Architecture (HPCA)*, 2010.

[21] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. 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.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of high performance computer architecture (HPCA)*, 2008.

[23] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2004.

[24] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[25] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proceedings of High-Performance Computer Architecture (HPCA)*, 2003.

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

[27] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *SIGARCH Computer Architecture News*, 35(2):57–68, 2007.

[28] 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 Symposium on Microarchitecture (MICRO)*, 2006.

[29] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *Proceedings of Parallel architectures and compilation techniques (PACT)*, 2006.

[30] T. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[31] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3:460–471, September 2010.

[32] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the international conference on Supercomputing (ICS)*, 1999.

[33] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proceedings of the Symposium on Microarchitecture (MICRO)*, 2008.

[34] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(1):135–144, 2008.

[35] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of Conference on Parallel and Distributed Computing Systems*, 2001.

[36] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[37] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Proceedings of the symposium on Microarchitecture (MICRO)*, 1997.

[38] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS)*, 2008.

[39] Vmware. `http://www.vmware.com/`.

[40] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev.*, 44:19–29, December 2010.

[41] Windows Azure Platform. `http://www.microsoft.com/windowsazure/`.

[42] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[43] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. *SIGARCH Computure Architecture News*, 34(2):155–166, 2006.

[44] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, 2006.

[45] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the European conference on Computer systems (EUROSYS)*, 2009.

[46] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In *Proceedings of the conference on Parallel architectures and compilation techniques (PACT)*, 2007.

[47] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Notices*, 45(3):129–142, 2010.