Optimizing Datacenter Power with Memory System Levers for Guaranteed Quality-of-Service [*]

Kshitij Sudan
University of Utah
kshitij@cs.utah.edu

Sadagopan Srinivasan
Intel Corporation
sadagopan.srinivasan@intel.com

Rajeev Balasubramonian
University of Utah
rajeev@cs.utah.edu

Ravi Iyer
Intel Corporation
ravishankar.iyer@intel.com

ABSTRACT

Co-location of applications is a proven technique to improve hardware utilization. Recent advances in virtualization have made co-location of independent applications on shared hardware a common scenario in datacenters. Co-location, while maintaining Quality-of-Service (QoS) for each application is a complex problem that is fast gaining relevance for these datacenters. The problem is exacerbated by the need for effective resource utilization at datacenter scales. In this work, we show that the memory system is a primary bottleneck in many workloads and is a more effective focal point when enforcing QoS. We examine four different memory system levers to enforce QoS: two that have been previously proposed, and two novel levers. We compare the effectiveness of each lever in minimizing power and resource needs, while enforcing QoS guarantees. We also evaluate the effectiveness of combining various levers and show that this combined approach can yield power reductions of up to 28%.

Categories and Subject Descriptors

C.0 [**GENERAL**]: System architectures, Hardware/software interfaces; C.4 [**PERFORMANCE OF SYSTEMS**]: Design studies; B.8.2 [**HARDWARE**]: Performance Analysis and Design Aids

Keywords

Quality-of-Service, datacenter power consumption, memory system architectures, Service Level Agreements (SLAs), datacenter management, application co-location or consolidation, cloud computing.

1. INTRODUCTION

Large datacenter facilities hosting customer applications provide performance guarantees as dictated by service level agreements (SLAs). With recent advances in cloud computing services, Quality-of-Service (QoS) guarantees are now an integral component of these SLA agreements. Customers are guaranteed the level of performance as defined by the resources they pay for. For example, customers "rent" CPU hours, RAM capacity, storage capacity, network bandwidth, etc. Each customer server is then an instantiation of a virtual machine (VM) with these allocated resources. Each VM in turn is hosted on a server and often co-located with other VMs. While co-locating different VMs on the same hardware, the service providers make sure that QoS guarantees are not violated by the co-location schedule. This typically leads to conservative schedules that do not necessarily make the most efficient use of the hardware resources.

Co-location of applications is not a new technology [11]. However, co-location in datacenters has been significantly boosted by recent advances in VM performance [3], and the ability migrate VMs across servers [5, 31]. Owing to the scale of these datacenters, efficient hardware utilization is a critical design point. Co-location of VMs/applications to increase server utilization is fast becoming the preferred approach to maximize utilization at these datacenters. For datacenters hosting customer applications instead of VMs, application co-location across servers is the equivalent of VM co-location. Application and VM migration has been extensively researched in the past [18, 27] and well known mechanisms for migration exist [9, 32]. To simplify the discussion (and subsequent evaluation) from here on, we will discuss our proposals in terms of co-located applications only.

A major concern while co-locating applications is whether any of the simultaneously executing applications on a server will violate their QoS guarantees. It is not acceptable to have a breach of QoS guarantee even if the server utilization improves. As a result, applications are typically scheduled conservatively leading to resource under-utilization [8]. This is an important problem for large datacenters where efficient use of hardware is a primary goal for profitable operation. In this work, we provide a framework that maximizes resource utilization, while maintaining QoS guarantees.

A typical server consolidation framework [1, 30] is expected to incorporate several complex features, most notably, to begin tasks on an appropriate server, to monitor the execution of programs, and to take corrective action if an application

[*] This work was supported in parts by NSF grants CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, and the University of Utah.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
Copyright 2012 ACM 978-1-4503-1182-3/12/09 ...$15.00.

threatens to not meet QoS constraints. In this evaluation, we will not model the dynamic adaptation aspects of the above framework, especially since simulators limit our ability to observe behavior for longer than a few seconds. We will instead focus on the process of starting a series of new tasks and identifying the maximum level of server consolidation that can be achieved. In other words, we will attempt to co-schedule N applications on to S servers, and devise techniques that allow us to minimize S, while ensuring that QoS guarantees are preserved. This allows us to power down as many servers as possible to reduce overall power[1].

In our evaluation, we attempt to incrementally pack each program on to an existing powered-up server. If this assignment disrupts the QoS guarantees for some of the applications (*sufferers*) on that server, we attempt to change various micro-architectural priorities to allow the sufferers to boost their throughput and meet their QoS constraint. We consider a number of such micro-architectural priority mechanisms and apply them incrementally until QoS is achieved or we exhaust all available mechanisms. In case of the latter, the suffering applications are then moved to a newly powered-up server. The ultimate metric of interest is the number of servers S activated after all the N programs have been successfully assigned, as to a first order, this has the biggest impact on overall power consumption.

We consider four different micro-architectural priority mechanisms in this work. In essence, each of these mechanisms allows restricting the resources allocated to non-suffering applications. By restricting resources of non-sufferers, more resources are available exclusively to sufferers to meet their QoS guarantees. With such a scheme, the suffering application(s) can meet its QoS constraint while hopefully not causing other applications to miss their constraints. Two of these mechanisms (b, c) have been proposed in prior work: (b) bandwidth priority [14] allocates memory bandwidth among competing programs by setting priorities in the memory controller scheduler, and (c) CacheScouts [33] can be used to allocate shared cache capacity among competing applications. The other two mechanisms (o, p) are being considered here for the first time: (o) open vs. closed page DRAM row buffer policy, where closed-page policy is employed for the non-suffering application, while open-page policy is employed for others, (p) prefetch, where aggressive memory prefetch is performed for the suffering application, while taking some bandwidth away from other applications.

We select these memory system levers because the memory system is known to be a major performance bottleneck in datacenters [4, 15, 16] and the memory system provides a rich set of options when varying application priority. Our study focuses on understanding the relative merits of these four mechanisms. We also attempt to understand how combinations of these mechanisms behave.

2. SHARED-RESOURCE PERFORMANCE

In this section we first present the impact of resource conflicts among concurrently executing applications on a chip multi-processor. Figure 1 shows the impact of co-locating applications on the same server. For this experiment, each application was first run on a 4-core server alone, and then with one other application, two other applications, and finally all four applications were run together. The system configuration is listed in Table 1. These results show that for enterprise class applications, performance degradation due to resource contention can vary from 17% (sap) to 33% (tpc) when all the cores of the CMP are utilized.

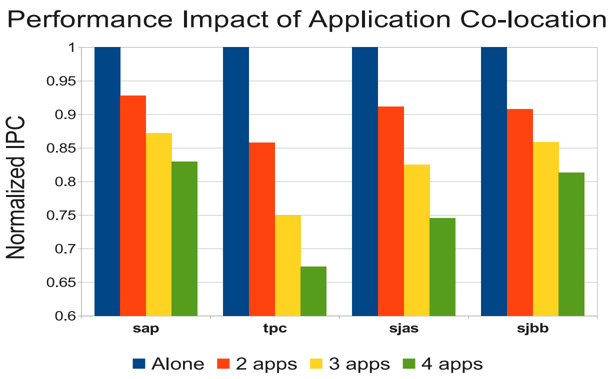


Figure 1: Impact on application performance due to co-location.

This degradation in performance is due to resource contention at various levels of the CMP. Fig. 2 shows the contention for L2 cache and impact on prefetcher's usefulness due to more than one application executing simultaneously on the 4-core server. Fig. 2(a) shows the change in L2 hit-rate. The L2 hit-rates reduce significantly for all application mixes due to higher conflict misses in the shared cache. Similarly, the usefulness of cache lines prefetched into the L2 also decreases as shown by Fig. 2(b). Most of this reduction in usefulness is due to the eviction of prefetched lines from the cache before they are accessed by the CPU.

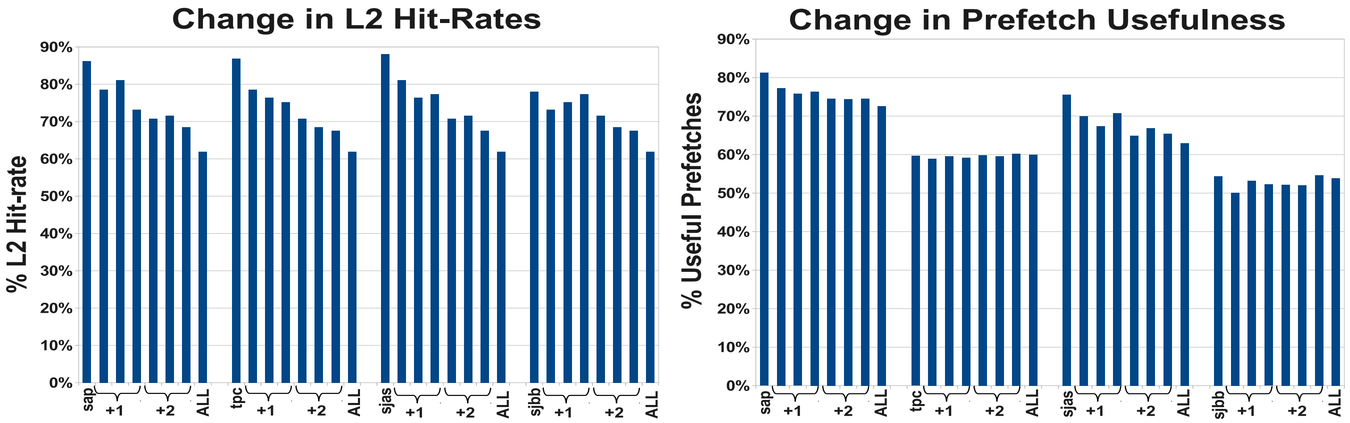
Similarly, Figure 3 shows the impact of application interference on total DRAM access cycles and DRAM row-buffer hit-rates. DRAM access cycles are counted only when the out-of-order core is stalling on a DRAM request. Fig. 3(a) shows that the number of DRAM cycles increases when more applications are executing because of an increase in queuing delays and DRAM service time. The decrease in row-buffer hit-rates shown in Fig. 3(b) contributes to this increase in DRAM access cycles. The row-buffer hit-rates in Fig. 3(b) are reduced when more than one application is executing because the CPU is now generating conflicting DRAM access requests from different applications.

3. HIGH-LEVEL QOS FRAMEWORK

To set the right context for our proposed mechanisms, we first describe the high-level overview of a QoS framework in future servers. Much of this framework is derived from existing designs [1, 10, 13, 23, 28, 30]. We will refer to this framework as the *High-Level Manager* or *HLM*. The SLAs serve as inputs to the HLM, specifying the expected performance metric (say, throughput, or IPC) for each application. The HLM executes in privileged mode and follows a "measure-check-reorganize" cycle to enforce QoS:

- "measure" refers to hardware performance counters that track the current performance level for each application.

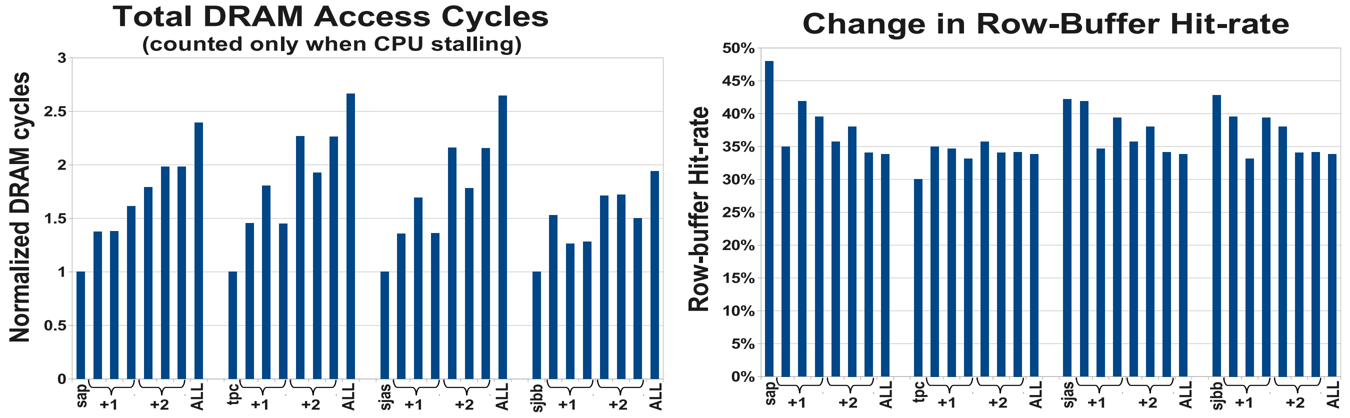
[1] It is almost always more power-efficient to overload a few servers than to distribute load across many servers because of leakage and other constant power delivery overheads [16, 25].



(a) Change in L2 cache hit-rate due to application interference.

(b) Change in usefulness of prefetched entries due to interference.

Figure 2: Impact of application interference on L2 hit-rate and prefetcher usefulness.



(a) Increase in total cycles spent accessing DRAM due to interference.

(b) Change in DRAM row-buffer hit-rate due to interference.

Figure 3: Impact of application interference on DRAM access cycles and row-buffer hit-rate.

- "check" refers to the HLM reading these performance counters at periodic intervals (referred to as *epochs*) and ensuring that QoS guarantees are being met, or if there is room to improve server utilization.
- "reorganize" refers to the HLM generating a new co-location schedule and performing application migration to meet QoS guarantees or improve server utilization.

When a new task arrives, the HLM attempts to schedule it on one of the already-active servers. Obviously, it is intractable for the HLM to evaluate every possible schedule to determine a power-optimal co-location. The HLM therefore relies on heuristics. In our evaluation, we consider two heuristics: (i) Low Load First (LLF): The new task is scheduled on the least loaded server. If QoS constraints are not met, the new task is migrated to a newly powered-up server. (ii) Heavy Load First (HLF): The new task is first scheduled on the most highly loaded server. If QoS is not achieved, the new task is migrated to the next highly loaded server, and so on. If no existing server can host the new task, it is moved to a newly powered-up server. The HLF policy may incur higher migration overheads, but it increases the likelihood that some lightly loaded server will exist to accommodate a resource-intensive task that may show up. More sophisticated task assignment policies may prune the search space by using pre-computed profiles of the application to help rule out some servers that may not have a light enough load.

Now consider the process of a new task assignment in more detail. After the new task is assigned to the least loaded server (in an LLF policy), we allow a few epochs to go by and then measure performance over the next few epochs. If QoS is not being met for some subset of applications on the server (referred to as the *sufferers*), we attempt to boost their performance by employing our micro-architectural priority levers to *non-suffering* applications. One lever is applied at a time and performance is measured for a few epochs. If QoS failure continues, more levers are applied. Some levers can have parameters associated with them that determine the extent of resource constriction for a non-sufferer. We limit ourselves to parameters that afford three different priority levels for any lever – low, medium, high. When activating a new lever, we first use the low priority level before mov-

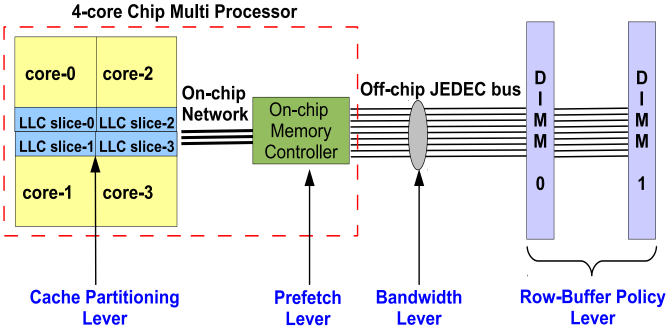


Figure 4: Memory Hierarchy Levers.

ing to medium and high. If we apply a lever change (say, we move from lever-a-low to lever-a-medium) and a hitherto non-suffering application joins the list of sufferers, we immediately cancel that lever change and move to the next item in the list of levers (lever-b-low). Once all the levers are exhausted for all non-suffering applications and QoS guarantees are still not achieved for suffering applications, we simply cancel all levers and move the new task to another server (either a new server (LLF) or the next highly loaded server (HLF)).

In summary, there are aspects of the HLM that are beyond the scope of this paper and are not modeled in our simulation framework (including re-assignment on phase changes, handling disruptions with conservative guarantees, LVT curves, etc.). The following aspects are modeled in the simulator:

- An input SLA specifies N tasks and corresponding QoS specifications. The QoS specification for each task is expressed as a number between 0 and 1, where 1 denotes the throughput of the task (represented by instructions per cycle or IPC) when it executes by itself on an unloaded server.

- Every few epochs, a new task from this list is selected for co-location. It is assigned to an existing server using either LLF or HLF.

- After some warm-up, measurements are taken over subsequent epochs to see if QoS guarantees are being met. If yes, then appropriate co-scheduling is achieved for the new task. Now wait till a new task is admitted.

- If QoS guarantees are not being met for some suffering applications, progressively move through the available set of levers. A lever can potentially have different strength levels. Levers (and their levels) are applied only if they do not introduce new applications into the list of sufferers.

- If all the levers fail to provide QoS, we move the new task to the next server (a newly powered-up server for LLF and the next highly loaded server for HLF).

- Once all tasks have been scheduled, the number of powered-up servers S represents the "goodness metric" for the server consolidation process.

4. MEMORY HIERARCHY LEVERS

Having described the overall QoS framework, we now turn our attention to the micro-architectural levers that allow us to prioritize a set of applications over other co-scheduled applications. These levers are employed when a set of applications (sufferers) are unable to meet their QoS constraints. The one common theme in all of these levers is that they introduce a trade-off between co-scheduled applications, *i.e.*, applying a lever for an application constricts the resources available to it, thereby boosting the performance of a *suffering* application by allocating more resources for its exclusive use. The levers considered in this study are shown in Figure 4. We first describe the two memory hierarchy levers (b and c) that have been employed in prior work.

- **(b) Bandwidth management:** We employ the priority mechanisms introduced by Iyer et al. [14] to provide higher memory bandwidth for suffering applications. With this lever, memory requests are scheduled by the memory controller based on priority ratios set by the HLM. If the HLM has set a priority ratio of 4:3, 4 high priority transactions for suffering applications are handled before 3 low priority transactions for non-sufferers. This is referred to as a bandwidth differential scheme. These priorities are implemented within the memory controller and the baseline memory controller uses the First-Ready-First-Come-First-Served (FR-FCFS [26]) scheduling policy that takes advantage of open rows in DRAM. When the bandwidth lever is applied, the scheduling policy becomes Priority-First-FR-FCFS, where we first apply the above bandwidth differential constraint before using FR-FCFS to select among remaining candidates. The strength of this lever is easily set by picking different bandwidth differential ratios.

- **(c) Cache allocation within the shared LLC:** To control the amount of shared cache capacity an application can use exclusively, we use the CacheScouts [33] mechanism. When this lever is employed, it acts to limit the cache capacity allocated to a given core. It does so by restricting the core to only occupy a maximum of N ways in any given cache set. When this core suffers a cache miss, if the number of lines occupied by the core in the incoming cache line's set is less than N, then the LRU line from the entire set is evicted. If however the core already occupies N lines in the set, then the LRU line among these N lines is evicted. The value of N depends on the applied level of this lever. If a core's cache capacity allocation is not being constrained by this lever, it evicts the LRU line from the incoming cache line's set, irrespective of the owner CPU.

Next, we discuss the two new memory-system levers:

- **(o) Open vs. closed page DRAM row buffer management policy:** Dynamically varying the row-buffer management policy can aid in QoS enforcement by allowing a new degree of control over access latency for DRAM requests. If an application is to be prioritized, we can prioritize it over the other by letting it use an open-row policy. This improves performance for the preferred application, while other applications are forced into using closed-row. Closing the row early for non-preferred applications also speeds up the next access from the preferred application. This lever can

CMP Parameters

ISA	x86-64 ISA	CMP size	4-core
L1 I-cache	128KB/2-way, private, 1-cycle	L1 D-cache	128KB/2-way, private, 1-cycle
L2 Cache	2 MB, 16-way, shared, 20-cycle	L1 and L2 Cache line size	64 Bytes

DRAM Parameters

DRAM Device Parameters	Micron MT47H64M8 DDR2-800 Timing parameters [17], t_{CL}=t_{RCD}=t_{RP}=20ns(4-4-4 @ 200 MHz) 4 banks/device, 16384 rows/bank, 512 columns/row, 32 bits/column, 8-bit output/device
DIMM Configuration	8 Non-ECC un-buffered DIMMs, 1 rank/DIMM, 64 bit channel, 8 devices/DIMM
DIMM-level Row-Buffer Size	32 bits/column × 512 columns/row × 8 devices/DIMM = 8 KB/DIMM
Active row-buffers per DIMM	4 (each bank in a device maintains a row-buffer)
Total DRAM Capacity	512 MBit/device × 8 devices/DIMM × 8 DIMMs = 4 GB

Table 1: Simulator parameters.

Application	LLC MPKI	DRAM access/Kilo-Inst
sap	1.91	3.71
tpc	8.23	13.25
sjas	4.77	9.49
sjbb	3.33	4.79

Table 2: Benchmark Properties. DRAM accesses include LLC misses and writebacks

Simulated Workload Mixes

Application	Workload Mix
Memory Intensive Mix (MI)	replicated copies of *tpc*
Balanced Mix (B)	replicated copies of *tpc, sap, sjas, sjbb*
Non-Memory Intensive Mix (NMI)	replicated copies of *sap*

Generating SLA Guarantees

SLA level	Fraction of baseline IPC
Relaxed	Random number between $[0.66, 0.77]$
Medium	Random number between $(0.77, 0.88]$
Aggressive	Random number between $(0.88, 1.0]$
Mix	Random number between $[0.66, 1.0]$

Table 3: Simulated Workload Mixes and SLA Guarantees.

be implemented with minor modifications to the memory controller architecture. The modification involves a small table at the memory controller which contains the process-ID and current row-buffer policy for each application executing on the CPU. This table is populated by the OS with inputs from the HLM. A page can also be closed after a timer expires and different levels of this lever can be implemented by using different timer values.

- **(p) Prefetch request scheduling:** We propose aggressive prefetch from DRAM as another lever. Whenever DRAM accesses are prefetched for an application, it introduces a trade-off since these applications now use more than their share of memory bandwidth to boost their performance. This lever is different than changing the DRAM bandwidth allocation since multiple prefetch requests are issued only when this lever is employed. This lever can be combined with the memory bandwidth lever to prevent low-priority applications from getting starved. We assume an integrated stream buffer plus memory controller. The operation of this integrated prefetch unit and memory controller is quite simple: if the access stream has locality and this lever is employed, the stream buffer issues prefetch requests for the next few cache lines after the current request. Low/medium/high versions of this priority mechanism can be constructed by tuning the threshold to detect streaming behavior and by issuing prefetches for fewer or more subsequent cache lines.

Most of the proposed hardware modifications are to the memory controller. Our design adds limited logic to adjust bandwidth priorities and adds more checks when scheduling accesses. The stream prefetcher has also been augmented. In addition to these changes, most requests must carry a process-ID as they propagate through the memory hierarchy. This allows the above levers to be employed only when applicable. The HLM relies on already existing hardware counters [2, 12] to track the relevant metrics in the SLA (IPC throughput for our evaluation).

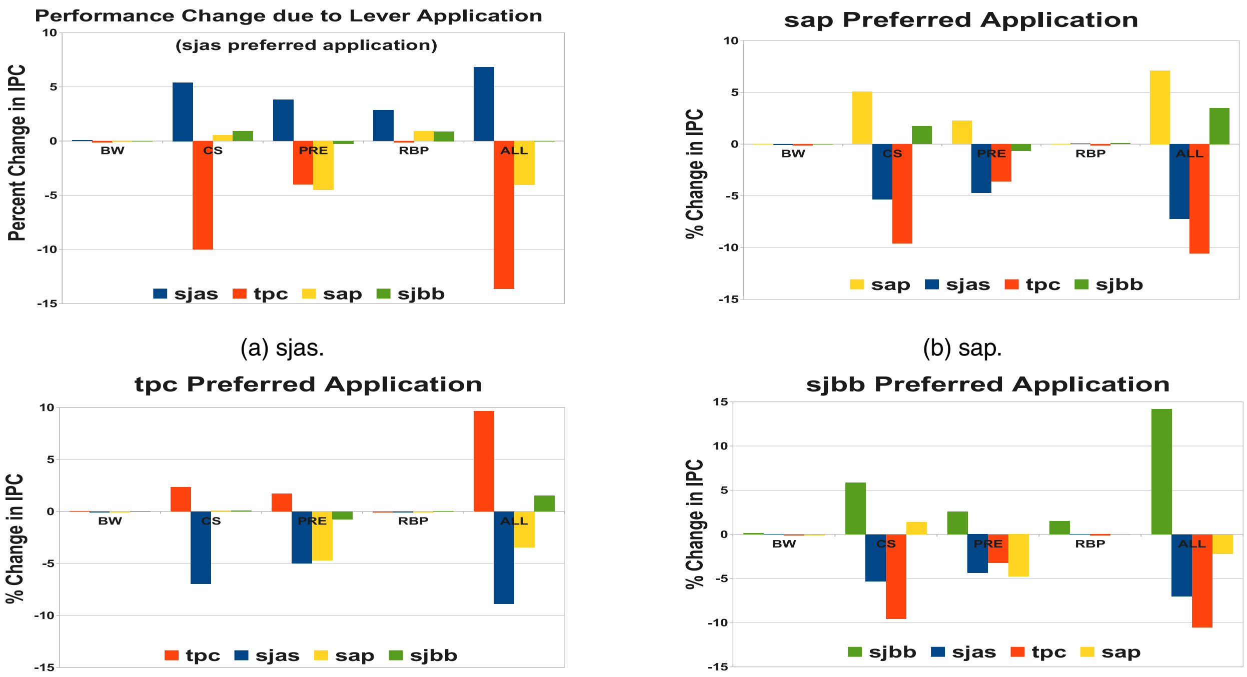
5. RESULTS

5.1 Methodology

To evaluate our proposals we use a trace-based simulator that processes traces collected from a real system. We focus only on single-threaded multi-programmed workloads to simplify the simulation. The simulator models an out-of-order CPU and models the execution of multi-programmed workloads on a Chip-Multiprocessor (CMP) platform. We model the memory hierarchy (including the memory controller and the DRAM sub-system) in detail and the parameters for the hierarchy are listed in Table 1.

The workloads chosen to simulate the system are the four enterprise benchmarks — *tpc, sap, sjas, sjbb.* The memory utilization metrics for these applications are listed in Table 2. A *job list* is used to denote the total number of jobs simulated for one experiment, and the order of applications admitted to the simulated datacenter. This job list is created by replicating the benchmarks traces an appropriate number of times. Specific *workload mixes* are created that signify the memory behavior of the job list. Job lists with three different memory access behaviors are used for our simulations and are listed in Table 3. When replicating for number of jobs greater than 4, the order of application admittance to the job pool was also randomized.

Finally, the customer specified input SLA guarantees are generated using a random number generator that specifies the fraction of IPC as the customer demanded QoS guarantee. We experimented with randomly generated SLAs between [0.66, 1] range - 0.66 signifying SLA guarantee equal to 66% of baseline IPC, and 1 signifying 100% of baseline performance of the application when executing alone on a 4-core CMP with parameters listed in Table 1. We did not consider SLA guarantees below 66% of baseline IPC because it is fairly easy for applications to meet this guarantee even when running with 3 other applications. Note that the choice of



(a) sjas.

(b) sap.

(c) tpc.

(b) sjbb.

Figure 5: Impact of individual lever application on application IPC.

using IPC as a metric for performance does not have any impact on the results. Any other performance metric, like transactions-per-second, can be used in place of IPC.

As listed in Table 3, four different SLA ranges are generated. Each range is meant to represent the application performance the customer expects. These guarantees are combined with the three workload mixes also listed in Table 3 to create one workload-SLA mix. There are thus a total of 12 such pairs (3 workload-mixes x 4 SLA guarantees). These 12 pairs represent various application mixes in terms of their memory intensiveness and SLA guarantees.

The power model for our work uses the following relation between the server utilization and total number of activated servers:

$\textit{Total power} = \textit{num servers} * (\textit{constant power draw} + \textit{server utilization} * (\textit{Watts/utility}))$

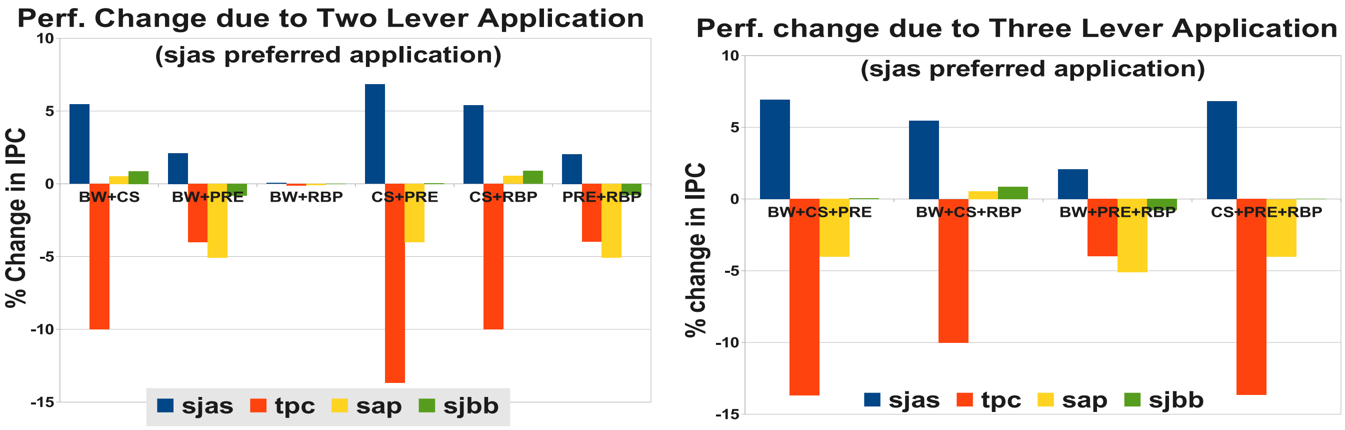
The $(Watts/utility)$ metric was obtained from Raghavendra et al.'s work [25] for a 2 GHz processor frequency (see Fig. 5 for Server B on page 5 [25]). This is essentially the slope of the line representing the 2 GHz frequency curve in the figure. Such data is consistent with that seen in other work [16]. Meisner et al. [16] show that a lightly loaded server consumes 270 W while a fully active server consumes 450 W. This data is the rationale behind the premise in Section 1 that it is better to consolidate applications to create few highly loaded servers than to create many lightly loaded servers.

5.2 Effectiveness of Levers in Controlling Application IPC

In this section we show results for how effective each lever is when employed to constrict resources for the non-suffering application on a server, and the corresponding increase in a suffering application's IPC. Fig. 5 shows the change in IPC of applications on a 4-core CMP server when different levers are employed. The first application is always the preferred application (the *suffering* application) and the other applications have their resources constricted by lever application. The X-axis shows the four different levers and when ALL levers are applied simultaneously. The Y-axis plots the percent change in IPC compared to the baseline when the application is executing alone on the CMP. As can be seen, the first application always shows improved IPC over baseline, while other three applications show reduction in IPC.

From these results, it can be observed that each lever (except BW) is able to change the IPC of applications to varying degrees. CS lever has the most impact followed by PRE lever, and then by RBP lever. This is to be expected since cache capacity is the most critical shared resource impacting performance. There is little change in IPC with BW lever because the benchmark applications do not saturate the provisioned system bandwidth. We expect other applications which consume more bandwidth than the current applications to show significant change in IPC with the BW lever. In Section 5.3 we provide sensitivity results for the bandwidth lever by artificially reducing system bandwidth. Another important trend to note is that when all levers are used collectively (ALL lever), the impact on IPC is cumulative of each lever's individual impact.

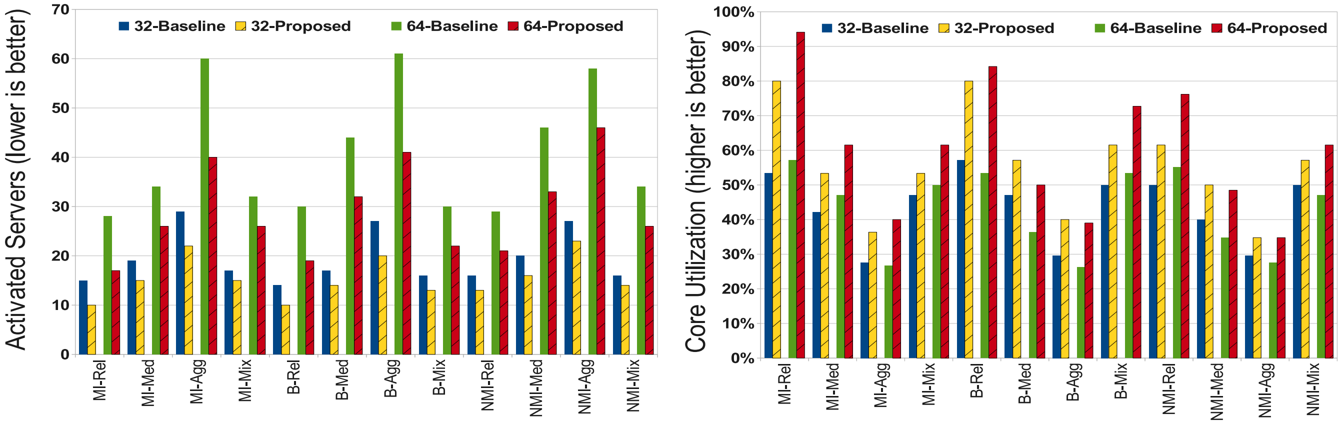
This trend of cumulative impact is examined in detail in Fig. 6(a) and (b) which show change in IPC when two, and three levers are employed respectively. It shows that lever application results in an additive impact on performance. Therefore, applying the levers in succession can improve sufferer application's IPC by progressively reducing available resources for the non-suffering applications. For these experiments, different lever combinations were applied to the



(a) Applying 2 levers simultaneously.

(b) Applying 3 levers simultaneously.

Figure 6: Impact of two and three simultaneous lever application on application IPC.



(a) Total servers activated.

(b) Activated core utilization.

Figure 7: Number of servers required, and percent of activated cores utilized with HLF policy.

applications, and as before, the first application is preferred. We show results only for *sjas* since other applications show similar trends.

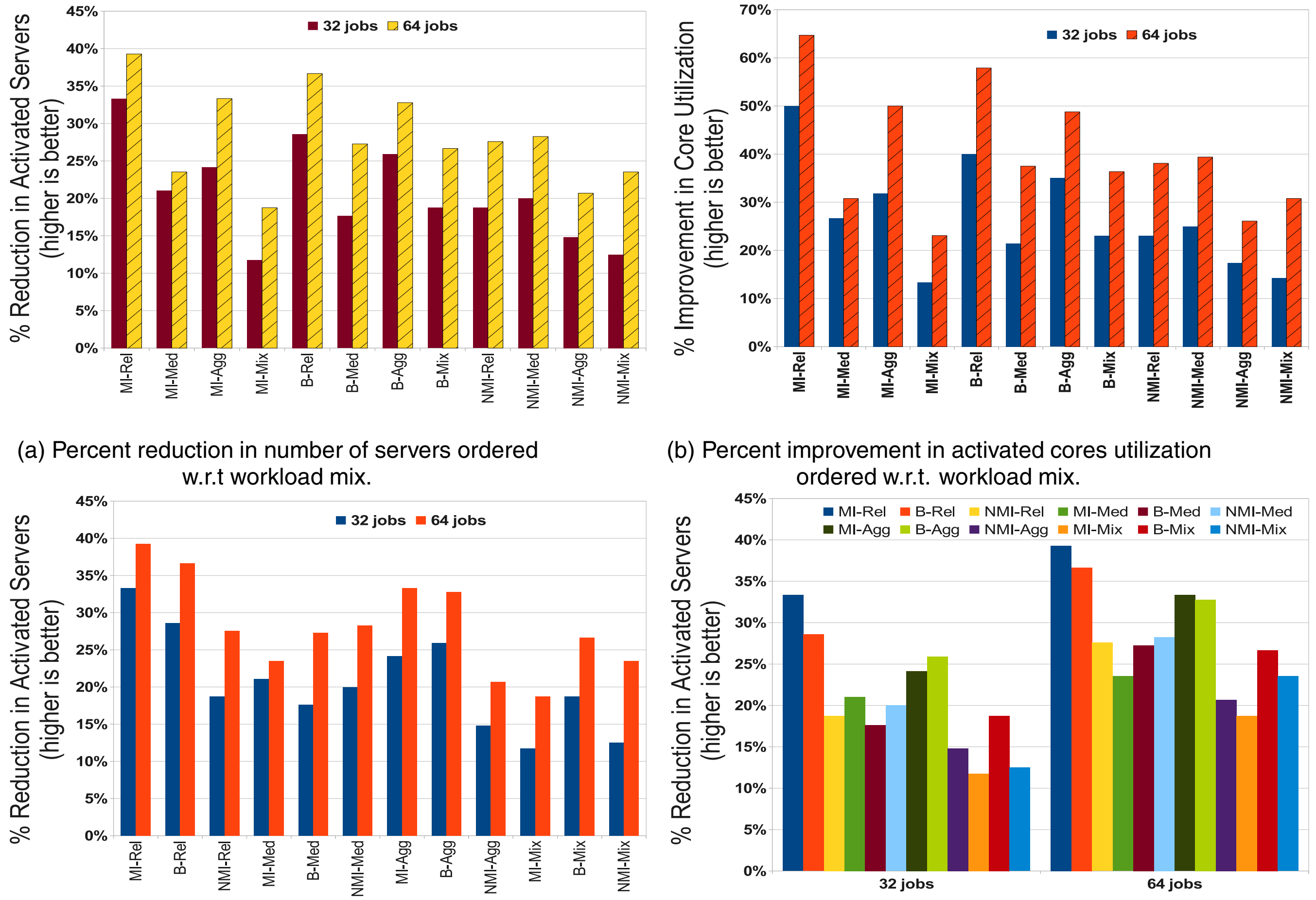
5.3 Activated Servers

In this section we study the impact of our proposed scheme on the metric of interest — N, the number of servers required. For these experiments, we varied the jobs allocated to the servers by creating workload mixes using the four applications. With these we try to understand the impact of workload mix, job list length, and SLA aggressiveness on the number of servers needed to execute all the applications while meeting their QoS. We compare our proposed results to a baseline, un-optimized system with no QoS levers.

Fig. 7 plots the number of activated servers and core utilization for the baseline and proposed schemes for the 12 workloads (3 benchmark mixes x 4 classes of SLAs). The job length is configured to 32, or 64 total jobs for these experiments. Core utilization shows what fraction of cores are being used among all the activated servers. This can be seen as a metric to measure the density of a co-location schedule with higher being better.

It can be seen that fewer servers need to be activated, and the utilization of activated cores increases when using the proposed scheme. Fig. 7(b) shows that the average core utilization for baseline is quite low — 43.6% and 42.9% for job length of 32 and 64 respectively. This shows that there is a heavy under utilization of the activated servers for the baseline system. With the proposed scheme the average core utilization increases to 55.4% and 60.4% respectively. These increases in utilization might seem small, but when multiplied by millions of cores at a typical datacenter, the efficiency gains are tremendous. Also, with the proposed scheme and longer job lengths, the core utilization will increase further due to reasons explained shortly.

To better understand the trends in Fig. 7, Fig. 8 re-plots the same data in a different way. Fig. 8(a) and 8(b) plot the percent reduction in activated servers and percent improvement in core utilization compared to the baseline. As the memory intensity of applications reduces from left to right, the improvements in core utilization and activated servers decreases. This is to be expected because the levers constrict memory resources and therefore have highest impact on memory intensive applications. For job length of 64, the average reduction in percent of activated servers for MI mix



(a) Percent reduction in number of servers ordered w.r.t workload mix.

(b) Percent improvement in activated cores utilization ordered w.r.t. workload mix.

(c) Percent reduction in server activation, ordered w.r.t. SLA aggressiveness.

(d) Percent reduction in server activation, ordered w.r.t. job list length.

Figure 8: Number of servers required, and percent of activated cores utilized with HLF policy.

is 30.8%, for Balance mix (B) is 28.7% and for NMI mix is 25%. With large datacenter installations, it is expected that the aggregate workload mix would resemble something like the Balanced mix in our simulations. Balanced mix shows improvements within 2% of MI mix.

Fig. 8(c) and Fig. 8(d) present the same data as in Fig. 7(a) but arranged differently to show how number of activated servers changes with respect to SLA aggressiveness and job list length. For Fig. 8(c), the SLA guarantees get more aggressive from left to right on the X-axis, and improvements also reduce. With more aggressive SLA, each application demands more resources. As a result, the ability to find applications that can be co-scheduled decreases resulting in reduced improvements.

In Fig. 8(d), when improvements are ordered with respect to length of the job list, a trend of improved performance emerges for longer job lists. This occurs due to the possibility of finding applications with a more diverse set of QoS requirements in a longer job list. We also experimented with job list lengths of 4, 8, and 16, and observed this trend of greater performance improvements with longer job lists in all our experiments.

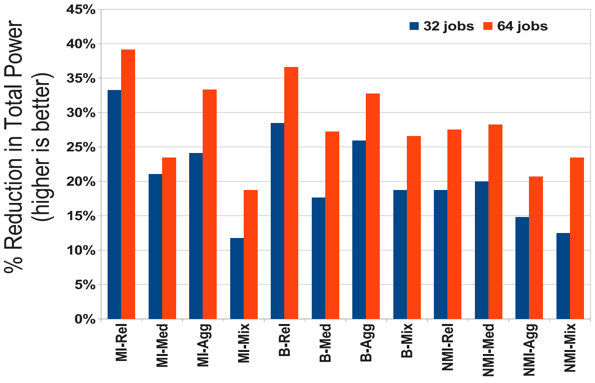


Figure 9: Percent reduction in total power.

Fig. 9 shows the percent reduction in power for the proposed scheme compared to the baseline. With increased improvements in server consolidation it shows that the total system power reduces in line with the improved consolidation for longer job lists. The average reduction in power for job list length of 32 is 20.5%, and 28.1% for list length of 64.

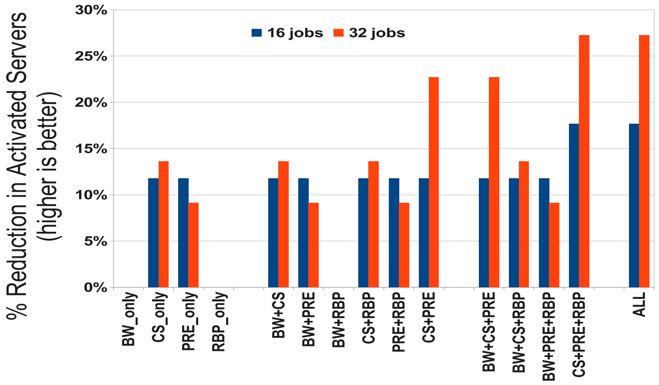


Figure 10: Improvement in activated servers when applying different combination of levers.

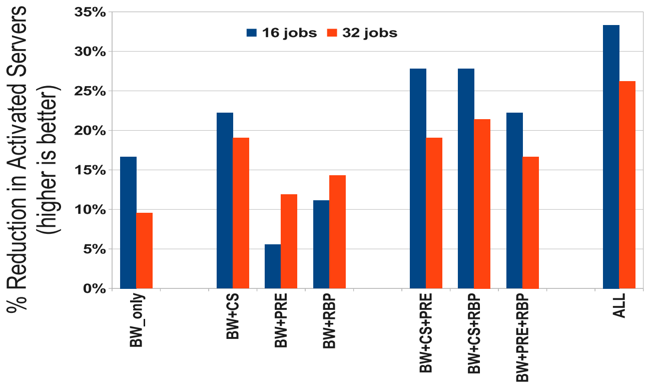


Figure 11: Sensitivity analysis for BW lever.

Fig. 10 shows the improvement in the number of activated servers compared to the baseline while applying varying number, and, varying combination of levers. Only the Balanced-Medium workload-SLA combination is shown in the graph for clarity. It can be observed that BW and RBP lever by themselves show no improvement. This is due to the fact that in our configured system, bandwidth is heavily under utilized. For the remaining combination of levers, we see a cumulative improvement in core utilization as expected. This graph can be correlated with Fig. 6 that shows that for our simulated workloads, the CS+PRE+RBP combination achieves all the improvements by managing the non-sufferer application's IPC and thus improving co-location density.

We expect the BW lever to be much more effective where the application performance is constricted by the aggregate system bandwidth. To perform a sensitivity analysis for the BW lever, we artificially reduced the system bandwidth by $4X$ and performed the same experiment as in Fig. 10, but only with lever combinations involving the BW lever. Fig. 11 shows that the BW lever is indeed effective at reducing the number of activated servers.

We also experimented with the Least-Loaded-First (LLF) job allocation policy described in Section 3. There were minor differences in the number of activated servers compared to the HLF policy. We do not present those results here due to space constraints.

6. RELATED WORK

Enforcing QoS guarantees has been an active area of research. The focus of most of the past work has been on making sure that no shared resource is exclusively over-consumed by a single application in the presence of other simultaneously executing applications. Some proposals like [8, 13, 14, 20, 33] have advanced frameworks that aim at improving overall throughput while enforcing these QoS guarantees. Each proposal has a different mechanism to enforce QoS at the architectural level, but they all are either at the shared cache level [33], or at the main memory bandwidth level [14]. Iyer et al. [14] propose a QoS enabled memory architecture that allocates shared cache capacity and memory bandwidth based on input from the operating environment. Guo et al. [8] use micro-architectural techniques that steal excess resources from applications while still enforcing QoS guarantees. They primarily focus on specification of QoS targets and the resulting job admission policy.

Another body of work [19, 21, 22] aims at improving resource utilization while providing QoS guarantees. These schemes enforce constraints at the DRAM request scheduling stage at the memory controller by modifying the order in which DRAM access commands are serviced. Nesbit et al. [21] propose "Private Virtual Time Memory System" for each thread to ensure each thread gets its allocated share of bandwidth irrespective of the system load. Rafique et al. [24] propose a simpler mechanism for fair allocation of bandwidth while avoiding bandwidth waste. Both these mechanisms only partition available memory bandwidth and do not mention how other resources interplay with such mechanisms. Mutlu and Moscibroda [19] make a critical observation that for chip multi-processor systems, inter-thread interference can introduce significant performance degradation. Their scheduler ensures that priority for open rows doesn't starve requests for closed rows.

The same authors improve upon their scheme by introducing "parallelism-aware batch scheduling" [22]. The idea is to process all possible DRAM requests in parallel for a thread (i.e., servicing requests which exhibit bank level parallelism), and in batches to provide QoS. These contributions are complementary to our proposal and would fall under the umbrella of memory controller levers (lever-(b) in our study). Ebrahimi et al. [6] show that to improve fairness in a CMP system, one needs to throttle the cores by limiting the requests they inject into the memory system. This differs from our proposal because we are not aiming at fairness of all executing applications on the CMP, instead we leverage the asymmetry among the application SLA guarantees to extract maximum resource usage from the server. The same authors also show in [7] that the prefetch requests can degrade system performance unless they are co-ordinated with other resource management policies. This aligns with our work as we show in Section 5 that the PRE lever is more effective when combined with other levers.

Recent work by Tang et al. [29] shows that for datacenter workloads, there is significant resource contention at all levels of the memory hierarchy. They do not consider interference at the DRAM level, but make the same conclusions as in Section 2 about interference at the last-level cache. Their work demonstrates the impact of application interference within caches on datacenter application performance. Their work differs from our proposal because we argue for implementing architectural levers that will help extract the maximum machine resource utilization while maintaining QoS for individual applications. We also show that there needs to be a co-ordinated approach while exercising these levers to achieve high co-location density.

7. CONCLUSIONS

Server consolidation at datacenters is essential for reducing power needs. In terms of power, it is beneficial to execute as many applications as possible on a single server. However, this runs the risk that some applications may not meet their QoS guarantees. To address this problem we study micro-architectural levers to improve co-location density. This work introduces two new levers: a row buffer management policy, and a prefetch lever, and two levers that have been previously proposed: cache and bandwidth allocation. We show that the effects of applying these levers is cumulative. The application of these levers can reduce the number of activated servers by up to 39%, resulting in a power reduction of up to 28%. Much future work remains in developing robust HLM features that can take advantage of these and additional micro-architectural levers.

8. REFERENCES

[1] Amazon Web Services. Amazon CloudWatch, Retrieved Oct. 2009.

[2] AMD Inc. BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors, Retrieved Oct. 2009.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of SOSP*, 2003.

[4] L. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of NSDI*, 2005.

[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A ConïňĄgurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Proceedings of ASPLOS*, 2010.

[7] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Prefetch-Aware Shared-Resource Management for Multi-Core Systems. In *Proceedings of ISCA*, 2011.

[8] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A Framework for Providing Quality of Service in Chip Multi-Processors. In *Proceedings of MICRO*, 2007.

[9] S. Hacking and B. Hudzia. Improving the Live Migration Process of Large Enterprise Applications. In *Proceedings of Workshop on Virtualization Technologies in Distributed Computing*, 2009.

[10] IBM Corporation. IBM Systems Director.

[11] IBM Corporation. IBM: z/VM Operating System.

[12] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2, Retrieved Oct. 2009.

[13] R. Iyer, R. Illikkal, L. Zhao, , D. Newell, and J. Moses. Virtual Platform Architectures: A Framework for Efficient Resource Metering in Datacenter Servers. Poster Session at SIGMETRICS, 2009.

[14] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of SIGMETRICS*, 2007.

[15] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. Reinhardt, and T. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of ISCA*, 2009.

[16] D. Meisner, B. Gold, and T. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of ASPLOS*, 2009.

[17] Micron Technology Inc. *Micron DDR2 SDRAM Part MT47H64M8*, 2004.

[18] D. Milojíčić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3), 2000.

[19] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of MICRO*, 2007.

[20] K. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proceedings ISCA*, 2007.

[21] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *Proceedings of MICRO*, 2006.

[22] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling - Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of ISCA*, 2008.

[23] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceeding of EuroSys*, 2007.

[24] N. Rafique, W. Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of PACT*, 2007.

[25] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *Proceedings of ASPLOS*, 2003.

[26] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory Access Scheduling. In *Proceedings of ISCA*, 2000.

[27] J. Smith. A Survey of Process Migration Mechanisms. *SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[28] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proceedings of HotPower - Workshop on Power Aware Computing and Systems*, 2008.

[29] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of ISCA*, 2011.

[30] VMware Inc. VMware vCenter, Retrieved Oct. 2009.

[31] VMware Inc. VMware VMotion, Retrieved Oct. 2009.

[32] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Process-Level Live Migration in HPC Environments. In *Proceedings of Supercomputing*, 2008.

[33] 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 PACT*, 2007.