A-DRM: Architecture-aware Distributed Resource Management of Virtualized Clusters

Hui Wang[†*], Canturk Isci[‡], Lavanya Subramanian[*], Jongmoo Choi[♮*], Depei Qian[†], Onur Mutlu[*]

[†]Beihang University, [‡]IBM Thomas J. Watson Research Center, [*]Carnegie Mellon University, [♮]Dankook University

{hui.wang, depeiq}@buaa.edu.cn, canturk@us.ibm.com, {lsubrama, onur}@cmu.edu, choijm@dankook.ac.kr

Abstract

Virtualization technology has been widely adopted by large-scale cloud computing platforms. These virtualized systems employ distributed resource management (DRM) to achieve high resource utilization and energy savings by dynamically migrating and consolidating virtual machines. DRM schemes usually use operating-system-level metrics, such as CPU utilization, memory capacity demand and I/O utilization, to detect and balance resource contention. However, they are oblivious to microarchitecture-level resource interference (e.g., memory bandwidth contention between different VMs running on a host), which is currently not exposed to the operating system.

We observe that the lack of visibility into microarchitecture-level resource interference significantly impacts the performance of virtualized systems. Motivated by this observation, we propose a novel architecture-aware DRM scheme (*A-DRM*), that takes into account microarchitecture-level resource interference when making migration decisions in a virtualized cluster. *A-DRM* makes use of three core techniques: 1) a profiler to monitor the microarchitecture-level resource usage behavior online for each physical host, 2) a memory bandwidth interference model to assess the interference degree among virtual machines on a host, and 3) a cost-benefit analysis to determine a candidate virtual machine and a host for migration.

Real system experiments on thirty randomly selected combinations of applications from the CPU2006, PARSEC, STREAM, NAS Parallel Benchmark suites in a four-host virtualized cluster show that *A-DRM* can improve performance by up to 26.55%, with an average of 9.67%, compared to traditional DRM schemes that lack visibility into microarchitecture-level resource utilization and contention.

VEE '15, March 14–15, 2015, Istanbul, Turkey.

ACM 978-1-4503-3450-1/15/03…$15.00.
http://dx.doi.org/10.1145/2731186.2731202

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling techniques, measurement techniques; D.4.8 [*Operating Systems*]: Performance – Modeling and prediction, measurements, operational analysis

Keywords virtualization; microarchitecture; live migration; performance counters; resource management

1. Introduction

Server virtualization and workload consolidation enable multiple workloads to share a single physical server, resulting in significant energy savings and utilization improvements. In addition to improved efficiency, virtualization drastically reduces operational costs through automated management of the distributed physical resources. Due to these attractive benefits, many enterprises, hosting providers, and cloud vendors have shifted to a virtualization-based model for running applications and providing various services (e.g., Amazon EC2 [2], Windows Azure [1]).

A key feature of virtualization platforms is the ability to move a *virtual machine (VM)* between physical hosts. This feature enables the migration of VMs to the appropriate physical hosts such that overall cluster efficiency is improved and resource utilization hot-spots are eliminated [15, 35, 43, 53]. In order to derive efficiency benefits from virtualization, the distributed resources should be managed effectively using an automated *Distributed Resource Management (DRM)* scheme. Such a scheme employs the VM migration feature judiciously to migrate VMs to the appropriate physical hosts such that VMs do not interfere with each other or significantly degrade each other's performance.

Many current DRM schemes [23, 27–31, 34, 56, 70, 72], including commercial products [31], manage VMs based solely on operating-system-level metrics, such as CPU utilization, memory capacity demand and I/O utilization. Such schemes do not consider interference at the microarchitecture-level resources such as the shared last level cache capacity and memory bandwidth. In this work, we observe that operating-system-level metrics like CPU utilization and memory capacity demand that are often used to determine which VMs should be migrated to which physical hosts cannot accurately characterize a workload's microarchitecture-level shared resource inter-

ference behavior. We observe that VMs may exhibit similar CPU utilization and memory capacity demand but very different memory bandwidth usage. Hence, DRM schemes that operate solely based on operating-system-level metrics could make migration decisions that leave the underlying microarchitecture-level interference unsolved or make it worse, leading to more interference at the microarchitecture-level and thus degrade the overall performance.

Some VM scheduling approaches (e.g., [75]) attempt to account for microarchitecture-level interference offline by employing a profiling phase to build a workload interference matrix that captures the interference characteristics when different pairs of applications are co-run. They then build constraints into the DRM scheme to forbid the co-location of workloads that heavily interfere with each other. We observe that such an *offline profiling* approach to incorporate microarchitecture-level interference into VM management at a cluster level has two major drawbacks. First, obtaining an interference matrix through profiling might not be feasible in all scenarios. For instance, it is not feasible for hosting providers and cloud vendors such as Amazon EC2 [2] and Microsoft Azure [1] to profile jobs from all users in advance, since it would incur prohibitive overhead to do so. Second, even if workloads can be profiled offline, due to workload phase changes and changing inputs, interference characteristics could change over execution. Hence, the interference matrix compiled from offline profiling might not accurately capture interference behavior during runtime.

Our goal, in this work, is to design a DRM scheme that takes into account interference between VMs at the microarchitecture-level shared resources, thereby improving overall system performance. To this end, we propose an architecture-aware DRM scheme, *A-DRM*. *A-DRM* takes into account two main sources of interference at the microarchitecture-level, 1) shared last level cache capacity and 2) memory bandwidth by monitoring three microarchitectural metrics: last level cache miss rate, memory bandwidth consumption and average memory latency. Specifically, *A-DRM* monitors the memory bandwidth utilization at each host. When the memory bandwidth utilization at a host exceeds a threshold, the host is identified as contended. Once such contention has been identified, the next key step is to identify which VMs should be migrated and to which hosts. In order to identify this, *A-DRM* performs a cost-benefit analysis to determine by how much each potential destination host's performance would be impacted if each VM on the contended host were migrated to it. Specifically, the cost-benefit analysis first estimates the increase in the last level cache miss rate at each potential destination host if each VM on the contended host were moved to it and then uses this miss rate increase to quantify the performance impact on the destination host.

We implement *A-DRM* on KVM 3.13.5-202 and QEMU 1.6.2, and perform comprehensive evaluations using a four-host cluster with various real workloads. Our experimental results show that *A-DRM* can improve the performance of a cluster by up to 26.55% with an average of 9.67%, and improve the memory bandwidth utilization by 17% on average (up to 36%), compared to a state-of-the-art DRM scheme [34] that does not take into account microarchitecture-level interference.

This paper makes the following contributions:

- We show that many real workloads exhibit different memory bandwidth and/or LLC usage behavior even though they have similar CPU utilization and memory capacity demand. Therefore, for effective distributed resource management, we need to consider not only operating-system-level metrics but also microarchitecture-level resource interference.
- We propose a model to assess the impact of interference on a host and perform a cost-benefit analysis to measure the impact of migrating every VM from a contended source host to a destination host.
- We propose *A-DRM*, which to our best knowledge, is the first DRM scheme that takes into account the characteristics of microarchitecture-level interference in making VM migration decisions, thereby mitigating interference and improving overall system (cluster) performance.
- We implement and evaluate our proposal on real hardware using diverse workloads, demonstrating significant performance improvements.
- We discuss several practical challenges that we encounter when implementing our *A-DRM* scheme, such as the effect of socket migration and the interconnection traffic.

2. Background and Motivation

In this section, we provide background on the main sources of microarchitecture-level interference. We then discuss the limitations of current DRM schemes that perform VM management using only operating-system-level metrics such as CPU utilization and memory capacity demand.

2.1 Microarchitecture-level Interference

The shared last level cache (LLC) capacity and main memory bandwidth are two major resources that are heavily contended between VMs sharing a machine [41, 42, 50, 51, 63]. Applications/VMs evict each other's data from the last level cache, causing an increase in memory access latency, thereby resulting in performance degradation [57, 58, 61]. Applications'/VMs' requests also contend at the different components of main memory, such as channels, ranks and banks, resulting in performance degradation [39, 41, 42, 47, 50, 51, 63, 64]. Different applications have different sensitivity to cache capacity and memory bandwidth [57, 63]. An application/VM's performance degradation depends on the application/VM's sensitivity to shared resources and the co-running applications/VMs [20, 63].

Today's servers typically employ a Non-Uniform Memory Access (NUMA) architecture, which has multiple sockets that are connected via interconnect links (e.g. QPI [33]). Several previous works propose to mitigate interference by migrating VMs *across* sockets such that applications/VMs

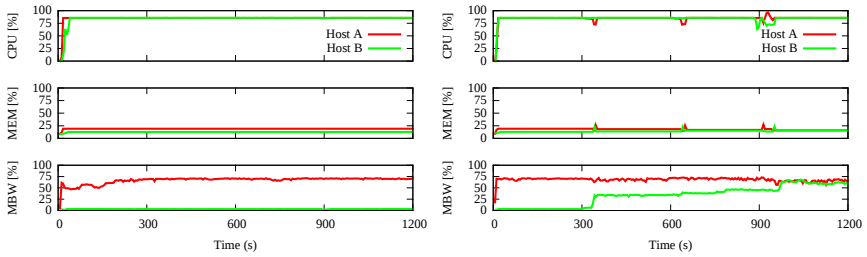


Figure 1: Resource utilization of Traditional DRM

Figure 2: Resource utilization of Traditional DRM + MBW-awareness

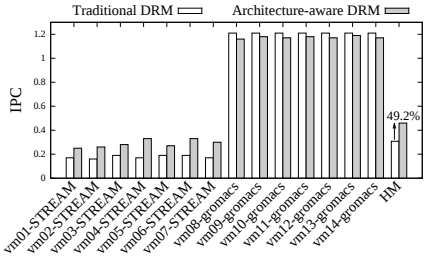


Figure 3: IPC Performance (HM is harmonic mean.)

that do not contend for the same shared resource are mapped to the same socket [12, 45, 60]. Our focus, in this work, is not on a single server, but on a *cluster of servers*. We explore VM migration across nodes, which is complementary to migrating applications/VMs across sockets.

2.2 Limitations of Traditional DRM Schemes

To address the VM-to-Host mapping challenge, prior works [23, 27–31, 34, 56, 72] have proposed to manage the physical resources by monitoring operating-system-level metrics (such as CPU utilization, memory capacity demand) and appropriately mapping VMs to hosts such that the utilization of CPU/memory resources is balanced across different hosts. While these schemes have been shown to be effective at CPU/memory resource scheduling and load balancing, they have a fundamental limitation – they are not aware of the microarchitecture-level shared resource interference.

2.2.1 Lack of Microarchitecture-level Shared Resource Interference Awareness

Prior works, including commercial products, base migration decisions on operating-system-level-metrics. However, such metrics cannot capture the microarchitecture-level shared resource interference characteristics. Our real workload profiling results (detailed in Section 6.1) show that there are many workloads, e.g., STREAM and gromacs, that exhibit similar CPU utilization and demand for memory capacity, but have very different memory bandwidth consumption. Thus, when VMs exhibit similar CPU and memory capacity utilization and the host is not overcommitted (i.e., CPU or memory is under-utilized), traditional DRM schemes that are unaware of microarchitecture-level shared resource interference characteristics would not recognize a problem and would let the current VM-to-host mapping continue. However, the physical host might, in reality, be experiencing heavy contention at the microarchitecture-level shared resources such as shared cache and main memory.

2.2.2 Offline Profiling to Characterize Interference

Some previous works [31, 37, 75] seek to mitigate interference between applications/VMs at the microarchitecture-level shared resources by defining constraints based on *offline profiling* of applications/VMs, such that applications that contend with each other are not co-located. For instance, in VMware DRS [31], rules can be defined for VM-to-VM or VM-to-Host mappings. While such an approach based on offline profiling could work in some scenarios, there are two major drawbacks to such an approach. First, it might not always be feasible to profile applications. For instance, in a cloud service such as Amazon EC2 [2] where VMs are leased to any user, it is not feasible to profile applications offline. Second, even when workloads can be profiled offline, due to workload phase changes and changing inputs, the interference characteristics might be different compared to when the offline profiling was performed. Hence, such an offline profiling approach has limited applicability.

2.3 The Impact of Interference Unawareness

In this section, we demonstrate the shortcomings of DRM schemes that are unaware of microarchitecture-level shared resource interference with case studies. We pick two applications: gromacs from the SPEC CPU2006 benchmark suite [6] and STREAM [7]. STREAM and gromacs have very similar memory capacity demand, while having very different memory bandwidth usage: STREAM has high bandwidth demand, gromacs has low (more workload pairs that have such characteristics can be found in Section 6.1).

We run seven copies (VMs) of STREAM on Host A and seven copies (VMs) of gromacs on Host B (initially). Both of the hosts are SuperMicro servers equipped with two Intel Xeon L5630 processors running at 2.13GHz (detailed in Section 5). Each VM is configured to have 1 vCPU and 2 GB memory.

Figure 1 shows the CPU utilization (CPU), total memory capacity demand of VMs over host memory capacity (memory capacity utilization - MEM), and memory bandwidth utilization (MBW) of the hosts when a traditional DRM scheme, which relies on CPU utilization and memory capacity demand, is employed. We see that although the memory bandwidth on Host A is heavily contended (close to achieving the practically possible peak bandwidth [21]), the traditional DRM scheme does nothing (i.e., does not migrate VMs) since the CPU and memory capacity on Host A and Host B are under-utilized and Host A and Host B have similar CPU and memory capacity demands for all VMs.

Figure 2 shows the same information for the same two hosts, Host A and Host B. However, we use a memory-bandwidth-contention-aware DRM scheme to migrate three VMs that consume the most memory bandwidth from Host A to Host B at 300 seconds, 600 seconds and 900 seconds. To keep the CPU resources from being oversubscribed, we also migrate three VMs that have low memory bandwidth requirements from Host B to Host A. We see that after the three migrations, the memory bandwidth usage on Host A

and Host B are balanced, compared to when employing the traditional DRM scheme (Figure 1).

Figure 3 shows the performance comparison between the traditional DRM and memory-bandwidth-contention-aware schemes, measured in IPC (Instructions Per Cycle). We see that the IPC of the VMs running STREAM increases dramatically (close to 2x in some cases). The harmonic mean of the IPC across all VMs improves by 49.2%, compared to the traditional DRM scheme. These results show that traditional DRM schemes that base their migration decisions solely on CPU utilization and memory capacity demand could leave significant performance potential unharnessed.

Our goal, in this work, is to design a DRM scheme that considers microarchitecture-level shared resource interference when making VM migration decisions such that interference is mitigated and resource utilization and performance are improved in a virtualized cluster.

3. *A-DRM*: Design

In this section, we describe the design of *A-DRM*, our proposed distributed resource management scheme that incorporates awareness of microarchitecture-level shared resource interference.

3.1 Overview

Figure 4 presents an overview of *A-DRM*, which consists of two components: a *profiler* deployed in each physical host and a *controller* that is run on a dedicated server.

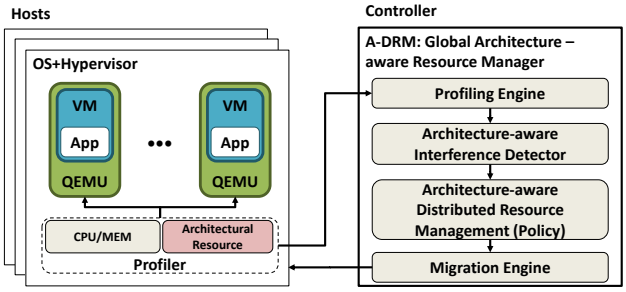


Figure 4: Prototype implementation

The primary objective of the *profiler* is to monitor resource usage/demands and report them to the *controller* periodically (at the end of every *profiling interval*). The *profiler* consists of two main components: 1) a CPU and memory profiler, which interacts with the hypervisor to get the CPU utilization and memory capacity demand of each VM and 2) an *architectural resource* profiler that leverages the hardware performance monitoring units (PMUs) to monitor the last level cache (LLC) capacity and memory bandwidth (MBW) usage of each VM. The *architectural resource* profiler also monitors the total memory bandwidth utilization and the average DRAM read latency, at each socket, to be used in detecting and managing interference.

The *controller* is the centerpiece of our distributed resource management framework. It is designed to detect microarchitecture-level shared resource interference and leverage this information to perform VM migration. The *controller* consists of four components: 1) A *profiling engine* that stores the data collected by the *profiler*. In order to improve accuracy and robustness in profiling data, a sliding window mechanism is used to calculate the moving average and smooth the profiled statistics. 2) An *architecture-aware interference detector* is invoked at each *scheduling interval* to detect microarchitecture-level shared resource interference. It detects hosts whose memory bandwidth utilization is greater than a threshold and classifies such hosts as contended. 3) Once such interference is detected, the *architecture-aware DRM policy* is used to determine new VM-to-Host mappings to mitigate the detected interference. The *architecture-aware DRM policy* computes the increase in LLC miss rates at each potential destination host, if each VM on a contended host were to be moved to it. It uses these miss rate increases to quantify the cost and benefit, in terms of performance impact at the source and destination hosts for every <contended host, VM, potential destination> tuple. This cost-benefit analysis is used to determine the best VM-to-host mappings. 4) The *migration engine* is then invoked to achieve the new VM-to-Host mappings via VM migration. These migrations could be configured to happen automatically or with the approval of the administrator.

3.2 Profiling Engine

The *profiling engine* stores the data collected by the *profiler* to quantify LLC and memory bandwidth interference, such as memory bandwidth consumption and LLC miss rate. The list of the monitored performance events and how exactly these are employed to quantify LLC and memory bandwidth interference are described in Table 1 and Section 4.

3.3 Architecture-aware Interference Detector

The *architecture-aware interference detector* detects the microarchitecture-level shared resource interference at each host. As we discussed in Section 2.1, the LLC capacity and main memory bandwidth are two major sources of microarchitecture-level shared resource interference. When VMs contend for the limited LLC capacity available on a host, they evict each other's data from the LLC. This increases data accesses to main memory, thereby increasing memory bandwidth consumption and interference. Furthermore, VMs' requests also contend for the limited main memory bandwidth at different main memory components such as channels, ranks and banks. Since the impact of both cache capacity and memory bandwidth interference is an increase in memory bandwidth utilization, the *architecture-aware interference detector* uses the memory bandwidth consumed at each host to determine the degree of microarchitecture-level shared resource interference. It computes the memory bandwidth utilization at each host as

$$MBW_{util} = \frac{ConsumedMemoryBandwidth}{PeakMemoryBandwidth}. \quad (1)$$

When the MBW_{util} at a host is greater than a threshold, $MBW_{Threshold}$, we identify the host as experiencing interference at the microarchitecture-level shared resources. We

provide more details on how we measure memory bandwidth in Section 4.1.

3.4 Architecture-aware DRM policy

The *architecture-aware DRM policy* is at the core of our *controller* and is invoked at the beginning of each *scheduling interval*. In this section, we present the high level design of our *architecture-aware DRM policy*. We provide more implementation details in Algorithm 1 and Section 4.2. Our DRM policy employs a two phase algorithm to determine an alternate VM-to-host mapping that mitigates interference.

Algorithm 1 *A-DRM*'s memory bandwidth based VM migration algorithm

```
1:  Input: Metrics (Snapshot of the measured metrics of entire cluster)
2:  RecommendedMigrations ← null
3:
4:  /* First phase: find a set of migrations to mitigate memory bandwidth
    interference */
5:  for each MBWContendedHost src in the cluster do
6:      while MBW_util of src > MBW_Threshold do
7:          MaxBenefit ← 0
8:          BestMigration ← null
9:          for each VM v in src do
10:             for each host dst with MBW_util < MBW_Threshold do
11:                 Benefit ← Benefit_vm + Benefit_src
12:                 Cost ← Cost_migration + Cost_dst
13:                 if Benefit − Cost > MaxBenefit then
14:                     MaxBenefit ← Benefit
15:                     BestMigration ← migrate v from src to dst
16:                 end if
17:             end for
18:         end for
19:         if BestMigration ≠ null then
20:             RecommendedMigrations.add(BestMigration)
21:             Update Metrics to reflect BestMigration
22:         end if
23:     end while
24: end for
25:
26: /* Second phase: balance CPU and memory utilization */
27: for each CPU/MemoryCapacityContendedHost src in the cluster do
28:     while src still CPU or Memory Capacity contended do
29:         MinMBWRatio ← 1    /* 0 ≤ Ratio ≤ 1 */
30:         BestMigration ← null
31:         for each VM v in src do
32:             for each host dst with CPU_util < CPU_Threshold or
                MEM_util < MEM_Threshold do
33:                 MBWRatio ← the MBW ratio on dst after migration
34:                 if MBWRatio < MinMBWRatio then
35:                     MinMBWRatio ← MBWRatio
36:                     BestMigration ← migrate v from src to dst
37:                 end if
38:             end for
39:         end for
40:         if BestMigration ≠ null then
41:             RecommendedMigrations.add(BestMigration)
42:             Update Metrics to reflect BestMigration
43:         end if
44:     end while
45: end for
46: return RecommendedMigrations
```

In the first phase, we use a greedy hill-climbing technique to determine the best VM-to-host mapping with the goal of mitigating microarchitecture-level shared resource interference. For each host that is detected as contended by the *architecture-aware interference detector* (MBWContendedHost in Algorithm 1 line 5), we aim to determine a set of migrations that provides the most benefit, while incurring the least cost. The *Benefit (line 11)* is an estimation of the improvement in performance if a VM were migrated, for both the VM under consideration to be migrated ($Benefit_{vm}$) and the other non-migrated VMs at the source host ($Benefit_{src}$). The *Cost (line 12)* is an estimation of the migration cost ($Cost_{migration}$) and degradation in performance at each potential destination host due to the migration ($Cost_{dst}$). We present detailed descriptions of these costs and benefits in Section 4.2. We employ a non-aggressive migration scheme that i) only migrates the least number of VMs to bring the host's MBW_{util} under the $MBW_{Threshold}$ (line 6), and ii) does not migrate at all if a good migration that has greater benefit than cost cannot be identified (line 13). For each contended host, after we determine a migration that provides the maximum benefit, we will accordingly update the memory bandwidth demand of the corresponding dst/src hosts by adding/subtracting the VM's bandwidth demand (line 21). The result of this phase is a set of recommended migrations that seek to mitigate microarchitecture-level interference. While the recommended migrations from this phase tackle the problem of microarchitecture-level shared resource interference, they do not take into account CPU utilization and memory capacity demand. This is done in the second phase.

The second phase, which is similar to traditional CPU and memory demand based DRM, balances the CPU and memory capacity utilization across all hosts, preventing CPU/memory capacity from being overcommitted, while keeping the cluster-wide memory bandwidth utilization balanced. Only after both phases are completed will the recommended migrations be committed to the *migration engine*.

3.5 Migration Engine

The *migration engine* performs the migrations generated by the *architecture-aware DRM policy*. We design the *migration engine* to avoid unnecessary migrations. Specifically, our *migration engine* has the ability to identify dependencies among recommendations and eliminate avoidable migrations. For instance, if *A-DRM* issues two migrations VM_A: $Host_X \rightarrow Host_Y$ (migrate VM A from host X to Y) and VM_A: $Host_Y \rightarrow Host_X$ (migrate VM A from host Y to X)[1], the *migration engine* would not issue them, since the second migration nullifies the effect of the first migration. Furthermore, if *A-DRM* issues two migrations $VM_A : Host_X \rightarrow Host_Y$ and $VM_A : Host_Y \rightarrow Host_Z$, the *migration engine* will combine them into one: VM_A: $Host_X \rightarrow Host_Z$, thereby improving the efficiency of migrations. After such dependencies have been resolved/combined, the remaining recommended migrations are executed.

[1] This is possible because $Metrics$ are continuously updated based on recommended migrations. As a result, future recommended migrations may contradict past recommended migrations.

4. *A-DRM*: Implementation

We prototype the proposed *A-DRM* on KVM 3.13.5-202 [43] and QEMU 1.6.2 [5]. The host used in our infrastructure is a NUMA system with two sockets (Section 5). We use the Linux performance monitoring tool *perf* to access the hardware performance counters, and the hardware performance events we use are listed in Table 1. To estimate the CPU demand of a VM, we use the mechanism proposed by [34]. The memory capacity metrics of a VM are obtained via libvirt [3]. We describe the details of our memory bandwidth measurement scheme and cost-benefit analysis in Sections 4.1 and 4.2 respectively.

4.1 Memory Bandwidth Measurement in NUMA Systems

In a NUMA system, the host contains several sockets and each socket is attached to one or more DIMMs (DRAM modules). For each socket, we measure the memory bandwidth using hardware performance events UNC_QMC_NORMAL_READS and UNC_QMC_WRITES, which includes any reads and writes to the attached DRAM memory. Thus the bandwidth consumption of the socket can calculated as

$$ConsumedMemoryBandwidth$$
$$= \frac{\text{64B} \times (\text{UNC\_QMC\_NORMAL\_READS} + \text{UNC\_QMC\_WRITES})}{ProfilingInterval}$$

since each of these reads and writes access 64 bytes of data. This bandwidth consumption is used along with the peak bandwidth to calculate memory bandwidth utilization (MBW_{util}), as shown in Equation 1. A host is identified as experiencing contention for microarchitecture-level shared resources only when all sockets on a host have MBW_{util} greater than the $MBW_{Threshold}$. While it is possible that only some of the sockets in a host could be contended, in a NUMA system, such interference can usually be mitigated by migrating VMs across sockets [12, 19, 45, 59, 60], which is orthogonal to our proposal.

We also estimate the bandwidth for each VM using OFFCORE_RESPONSE (in Table 1), which tracks the number of all requests from the corresponding VM to the DRAM. The per-VM bandwidth metrics are correspondingly added/subtracted from the socket-level bandwidth utilization metrics to estimate the new memory bandwidth utilizations during the execution of Algorithm 1.

4.2 Cost-Benefit Analysis

The main objective of the cost-benefit analysis is to filter out migrations that do *not* provide performance improvement or that degrade performance. For a given migration tuple <*src*, *vm*, *dst*>, indicating migration of *vm* from host *src* to host *dst* the costs include: 1) the VM migration cost and 2) performance degradation at the destination host due to increased interference. The benefits include: 1) performance improvement of the migrated VM and 2) performance improvement of the other VMs on the source host due to reduced interference. To quantitatively estimate the costs and benefits, all four types of costs/benefits are modeled as time overheads.

Table 1: Hardware Performance Events

Hardware Events	Description
OFFCORE_RESPONSE	Requests serviced by DRAM
UNC_QMC_NORMAL_READS	Memory reads
UNC_QMC_WRITES	Memory writes
UNC_QMC_OCCUPANCY	Read request occupancy
LLC_MISSES	Last level cache misses
LLC_REFERENCES	Last level cache accesses
INSTRUCTION_RETIRED	Retired instructions
UNHALTED_CORE_CYCLES	Unhalted cycles

4.2.1 Cost: VM Migration

VM migration incurs high cost since all of the VM's pages need to be iteratively scanned, tracked and transferred from the host to the destination. *A-DRM* models the cost of VM migration by estimating how long a VM would take to migrate. This cost depends mainly on the amount of memory used by the VM, network speed and how actively the VM modifies its pages during migration.

The VM migration approach used in *A-DRM* is a *pre-copy*-based live migration [15, 43, 53] with timeout support. Initially, live migration (that does not suspend the operation of the VM) is employed. If the migration does not finish within a certain time (*live migration timeout*), *A-DRM* switches to an offline migration approach, which suspends the entire VM and completes the migration.

A-DRM calculates the time required for VM migration ($Cost_{migration}$ in Algorithm 1) based on the VM's current active memory size, the dirty page generation rate and the data transfer rate across the network.

$$Cost_{migration} = \frac{ActiveMemorySize}{DataTransferRate - DirtyRate}$$

4.2.2 Cost: Performance Degradation at *dst*

By migrating a VM to a host, the VM would compete for resources with other VMs on the destination host. The main sources of contention at the destination host are the shared LLC capacity and memory bandwidth. The VMs at the destination host would experience interference from the migrated VM at these shared resources, thereby stalling for longer times. The *Stall cycles* (*Stall* for short) indicates the latency experienced by a VM from waiting for requests to the LLC and DRAM, during the previous *scheduling interval*. It is calculated as:

$$Stall = NumLLCHits * LLCLatency +$$
$$NumDRAMAccesses * AvgDRAMLatency$$

We measure the $NumLLCHits$ as the difference between the performance events LLC_REFERENCES and LLC_MISSES (in Table 1). We use a fixed $LLCLatency$ in our system [32]. We use the performance event OFFCORE_RESPONSE to estimate $NumDRAMAccesses$. We estimate the $AvgDRAMLatency$ using performance events UNC_QMC_OCCUPANCY and

UNC_QMC_NORMAL_READS (in Table 1) as:

$$AvgDRAMLatency = \frac{\text{UNC\_QMC\_OCCUPANCY}}{\text{UNC\_QMC\_NORMAL\_READS}}$$

For every migration tuple <*src*, *vm*, *dst*>, *A-DRM* uses a simple linear model to estimate the new *Stall* of each VM on the destination host after the migration, as a function of last level cache misses per kilo cycles (MPKC), as follows:

$$NewStall_i = Stall_i \times \frac{MPKC_{dst} + MPKC_{vm}}{MPKC_{dst}}, \forall i \in dst$$

$MPKC_{vm}$ is the MPKC of the migrated *vm*, while $MPKC_{dst}$ is the sum of MPKCs of all VMs on the destination host. This simple linear model assumes that the *Stall* for each VM on the destination host, *dst*, increases linearly as the increase in MPKC (LLC miss rate) at the destination host, if *vm* were moved from *src* to *dst*.

The increase in stall time for each individual VM, i, on the destination host (from the linear model above) is

$$DeltaStall_i = \frac{Stall_i \times MPKC_{vm}}{MPKC_{dst}}$$

The overall cost (performance degradation) on the destination host, in terms of time overhead, of migrating *vm* to *dst* is calculated as the sum of the stall time increase of each VM:

$$Cost_{dst} = \sum_{i \in dst} DeltaStall_i$$

4.2.3 Benefit: Performance Improvement of *vm*

A similar linear model as the previous subsection can be used to model the performance benefit experienced by the migrated *vm*:

$$Benefit_{vm} = \frac{Stall_{vm} \times MPKC_{src}}{MPKC_{dst}}$$

$MPKC_{src}$ is the sum of the MPKCs of all VMs on the source host. The migrated *vm*'s stall time reduces/increases proportionally to the ratio of the source and destination's memory bandwidth demand (MPKC), using a linear model.

4.2.4 Benefit: Performance Improvement at *src*

The performance improvement experienced by the VMs remaining on the *src* host can be estimated as:

$$Benefit_{src} = \sum_{j \in src} \frac{Stall_j \times MPKC_{vm}}{MPKC_{src}}$$

The *Stall* of the remaining VMs on the source host reduces proportionally to the memory bandwidth demand (MPKC) of the migrated VM (*vm*), using our linear model.

5. Methodology

We conduct our experiments on a cluster of four homogeneous NUMA servers and a Network-Attached Storage (NAS). All servers and the shared NAS are connected via a 1 Gbps network. Each server is dual-socket with a 4-core Intel Xeon L5630 (Westmere-EP). Each core has a 32KB private L1 instruction cache, a 32KB private L1 data cache, a 256KB private L2 cache, and each socket has a shared 12MB L3 cache. Each socket is equipped with one 8GB DDR3-1066 DIMM. We disable turbo boost and hyper-threading to maximize repeatability. The hypervisor is KVM. The OS is Fedora release 20 with Linux kernel version 3.13.5-202. The QEMU and libvirt versions are 1.6.2 and 1.1.3.5, respectively. Each virtual machine is configured to have 1 vCPU and 2 GB memory.

Workloads. Our workloads are shown in Table 2. We use applications from the SPEC CPU2006 [6], PARSEC [11], NAS Parallel Benchmark [4] suites and the STREAM Benchmark [7, 47]. We also include two microbenchmarks: *MemoryHog* and *CPUUtilization*, which saturate the memory capacity and CPU resources, respectively. We first profile the microarchitecture-level shared resource usage of each application by running it alone inside a VM and pinning the CPU and memory to the same socket. We classify an application as memory-intensive if its memory bandwidth consumption is beyond 1 GB/s, and memory-non-intensive otherwise (details in Section 6.1). Except for the profiling experiments, applications are iteratively run inside VMs.

Metrics. We use the harmonic mean of Instructions Per Cycle (IPC) and weighted speedup [22, 62] to measure performance. We use the maximum slowdown metric [16, 41, 42, 69] to measure unfairness.

Comparison Points and Parameters. We compare *A-DRM* to a traditional CPU and memory demand based DRM policy [34] (*Traditional DRM* in short). We employ the same methodology to estimate the CPU utilization and memory capacity demand for *A-DRM* and *Traditional DRM*. When the total demand of all the VMs is beyond the host's capacity, it migrates the VMs to other under-utilized hosts. The parameters used in our experiments are summarized below:

Parameter Name	Value
CPU overcommit threshold ($CPU_{Threshold}$)	90%
Memory overcommit threshold ($MEM_{Threshold}$)	95%
Memory bandwidth threshold ($MBW_{Threshold}$)	60%
DRM scheduling interval (*scheduling interval*)	300 seconds
DRM sliding window size	80 samples
Profiling interval (*profiling interval*)	5 seconds
Live migration timeout (*live migration timeout*)	30 seconds

6. Evaluation

In this section, we present our major evaluation results. First, we present the characteristics of workloads, namely, memory bandwidth, LLC hit ratio and memory capacity demand. Afterwards, we revisit our motivational example (Section 2.3), demonstrating the resource utilization behavior and benefits with *A-DRM*. We then evaluate the effectiveness of *A-DRM* with a variety of workload combinations. Finally, we present sensitivity to different algorithm parameters and summarize the lessons learned from our evaluations.

6.1 Workload Characterization

Figure 5 and Figure 6 show the memory capacity demand, memory bandwidth and LLC hit ratio for each workload considered in this study (Note the CPU utilization for each

Table 2: Workloads

Suites	Benchmarks (55 total)	Memory Intensity
SPEC CPU2006	bwaves, mcf, milc, leslie3d, soplex, GemsFDTD, libquantum, lbm (8 total)	memory-intensive
	perlbench, bzip2, gcc, gamess, zeusmp, gromacs, cactusADM, namd, gobmk, dealII, provary, calculix, hmmer, sjeng, h264ref, tonto, omnetpp, astar, sphinx3, xalancbmk (20 total)	memory-non-intensive
PARSEC 2.1	streamcluster (1 total)	memory-intensive
	blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, swaptions, x264 (9 total)	memory-non-intensive
NAS Parallel Benchmark	cg.B, cg.C, lu.C, sp.B, sp.C, ua.B, ua.C (7 total)	memory-intensive
	bt.B, bt.C, ep.C ft.B, is.C, lu.B, mg.C (7 total)	memory-non-intensive
STREAM	STREAM (1 total)	memory-intensive
Microbenchmark	MemoryHog, CPUUtilization (2 total)	memory-non-intensive

VM is always above 95%, which is not shown in the figures). The reported values are measured when we run each workload alone and pin the vCPU and memory to the same socket. We make the following observations.

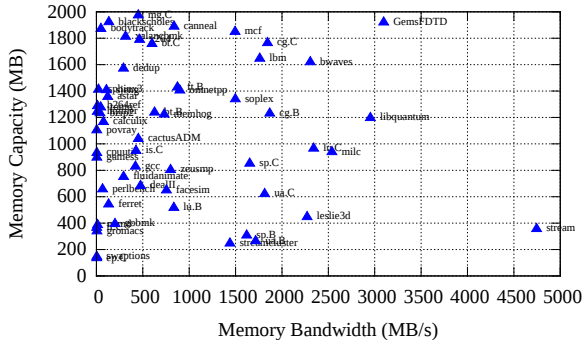


Figure 5: Memory capacity vs. memory bandwidth consumption

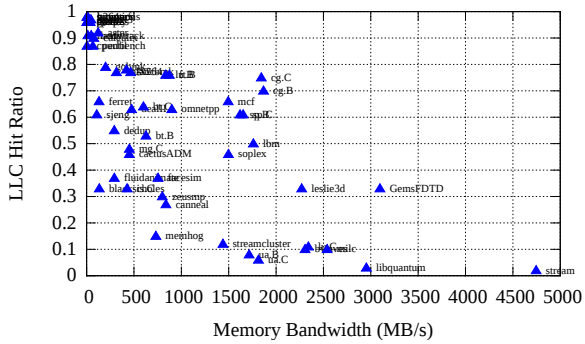


Figure 6: LLC hit ratio vs. memory bandwidth consumption

First, there is no strong correlation between memory capacity demand and memory bandwidth (Figure 5). For instance, the workloads STREAM and gromacs have similar memory capacity demand (around 400MB) while their memory bandwidth requirements are quite different (memory bandwidth requirement of STREAM is 4.5 GB/s while that of gromacs is 0.1 GB/s). There are more such pairs (e.g., blackscholes vs. GemsFDTD, perlbench vs. facesim), indicating that several workloads exhibit different memory bandwidth requirements, while having similar memory capacity demand.[2] Second, generally, workloads that consume low memory bandwidth exhibit a high LLC hit ratio (Figure 6). However, there exist cases where workloads consume very different amounts of memory bandwidth even when they exhibit similar LLC hit ratios (e.g., leslie3d and blackscholes). This is because the hit ratio only captures what fraction of accesses hit in the cache, whereas the absolute number of requests to the shared memory hierarchy could be very different for different applications. Third, when workloads that have high bandwidth requirements (often due to a low LLC hit ratio) are co-located on the same host, they tend to interfere (as they both access memory frequently), degrading performance significantly. To prevent this contention, we need to consider memory bandwidth and LLC usage behavior in making co-location decisions, as we describe in Equation (1) and Section 4.2.

[2] Similarly, we observed no correlation between the memory bandwidth demand and CPU utilization of a workload.

6.2 *A-DRM* Case Study

We revisit the example discussed in Section 2.3 and demonstrate the effectiveness of *A-DRM* on the workloads in the example. Figure 7 shows the impact (over time), of applying *A-DRM* on our workloads. At the bottom, the evolution of VM-to-host mappings is shown (labeled from Ⓐ to Ⓓ). There are two hosts and seven VMs on each host. Each VM runs either the *STREAM* benchmark which is represented as a rectangle with vertical lines (denoted as "H", meaning that it has high memory bandwidth demand) or the *gromacs* benchmark which is represented as a rectangle with horizontal lines (denoted as "L", meaning it has low memory bandwidth demand). The figure also shows the variation of the total CPU utilization (CPU), memory capacity demand utilization (MEM) and memory bandwidth utilization (MBW) (normalized to the total capacity of the host) on each of the hosts, as time goes by. The timeline is labeled from ① to ⑥.

In the initial state (labeled as Ⓐ), we configure all VMs (VM01-VM07) in Host A to run the *STREAM* benchmark, while all VMs (VM08-VM14) in Host B run the *gromacs* benchmark. Since *STREAM* has high memory bandwidth demand, Host A becomes bandwidth-starved, while Host B which executes *gromacs* (an application with low memory bandwidth demand) has ample unused bandwidth. At the end of the first *scheduling interval* (300 seconds, ①), *A-DRM* detects that the memory bandwidth utilization of Host A is above the $MBW_{Threshold}$ (60%).

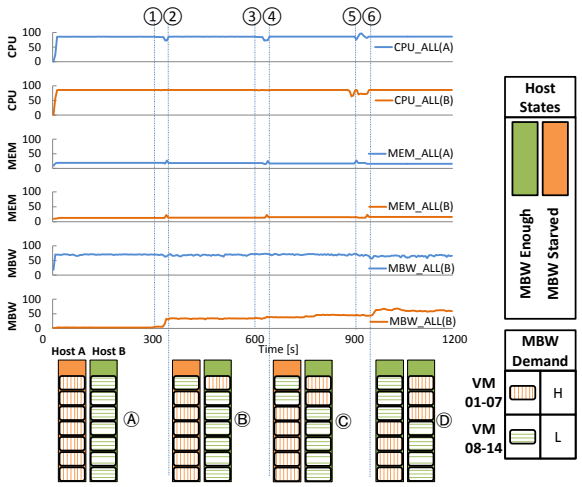


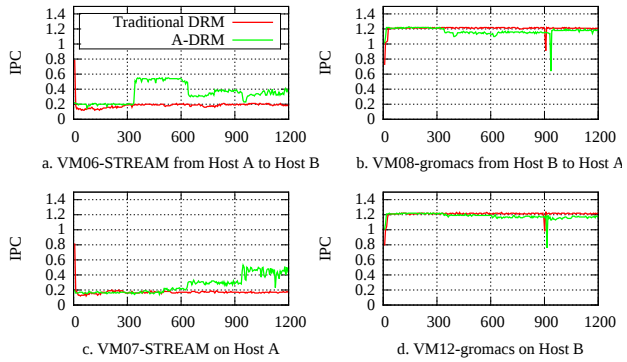
Figure 7: *A-DRM* execution timeline

The *architecture-aware DRM policy* is then invoked. Upon execution of the first phase of Algorithm 1, *A-DRM* selects one VM on Host A, which provides the maximum benefit based on our cost-benefit analysis and migrates it to Host B. Then, upon execution of the second phase of the algorithm, *A-DRM* selects a VM from Host B to re-balance the CPU utilization between the two hosts. As a result of the execution of both phases, the VM-to-host mapping changes, as shown in Ⓑ. Note that after time ②, when this migration is completed, the memory bandwidth usage of Host B increases due to the migrated VM.

Furthermore, it is important to note that while the bandwidth usage of Host B increases, the bandwidth usage of Host A does not decrease even though *gromacs* has lower memory-bandwidth usage than *STREAM*. This is because the freed up memory bandwidth is used by the remaining VMs. Hence, at the next scheduling point (600 seconds, ③), *A-DRM* detects bandwidth contention and invokes the two phases of the *architecture-aware DRM policy* and performs another migration. The result is the VM-to-host mapping in Ⓒ. This process then repeats again at the 900 second scheduling point (⑤), resulting in the VM-to-host mapping in Ⓓ. At this point, the memory bandwidth utilization of the two hosts are similar. Hence, the cost-benefit analysis prevents *A-DRM* from migrating any further VMs, since there is no benefit obtainable from such migrations.

Figure 8 shows the performance of four representative VMs from this experiment. VM06 and VM08 are initially located on Host A and Host B respectively. At the 300-second scheduling point in Figure 7, VM06 is migrated from Host A to Host B, while VM08 is migrated from Host B to Host A. VM07 and VM12 remain on Host A and Host B respectively and are not migrated.

Figure 8a presents the IPC for VM06, which runs *STREAM* and has high memory bandwidth demand. After migration (at 300 seconds), the IPC of VM06 increases significantly. At the 600-second scheduling point, VM06's performance degrades since another VM is migrated from Host A to Host B. As VMs are migrated away from Host A,



a. VM06-STREAM from Host A to Host B
b. VM08-gromacs from Host B to Host A
c. VM07-STREAM on Host A
d. VM12-gromacs on Host B

Figure 8: IPC of VMs with *A-DRM* vs. Traditional DRM

VM07's (which remains on Host A) performance (Figure 8c) improves significantly (at 600 and 900 seconds).

A potential concern is that the VMs running *gromacs* may experience performance degradation when co-located on the same host as *STREAM*. However, Figure 8b and Figure 8d show that VM08 , the VM migrated from Host B to Host A and VM12, the VM remaining on Host B experience only a slight drop in performance, since *gromacs* does not require significant memory bandwidth.[3] Therefore, we conclude that by migrating VMs appropriately using online measurement of microarchitecture-level resource usage, *A-DRM* alleviates resource contention, achieving better performance for contended VMs. Furthermore, the cost-benefit analysis prevents migrations that would not provide benefit.

6.3 Performance Studies for Heterogeneous Workloads

Our evaluations until now have focused on a homogeneous workload, to help better understand the insights behind *A-DRM*. In a real virtualized cluster, however, there exist a variety of heterogeneous workloads, showing diverse resource demands. In this section, we evaluate the effectiveness of *A-DRM* for such heterogeneous workloads.

Figure 9 presents the improvement in harmonic mean of IPCs (across all VMs in a workload) for 30 workloads, from our experiments on a 4-node cluster. We run 7 VMs on each node, making up a total of 28 VMs. Each VM runs a benchmark, which is selected randomly from Table 2. Each workload *i*X*n*Y-Z, consists of X VMs that run memory-*i*ntensive applications and Y VMs that run *n*on-intensive applications. We evaluate two different workloads for each intensity composition, the number of which is denoted by Z. For instance, *i*12*n*16-1 means that 12 VMs are memory-intensive and 16 VMs are memory-non-intensive (randomly selected from Table 2), and this is the first workload with this particular intensity composition.

We draw three main conclusions from Figure 9. First, *A-DRM* improves performance by up to 26.55%, with an average improvement of 9.67% across all our 30 workloads, compared to traditional DRM [34]. Second, *A-DRM* outperforms traditional DRM for *all* 30 workloads, indicating that microarchitecture-level interference-awareness is an impor-

[3] The IPC of VM08 and VM12 drops for a brief period around 900 second since we run gromacs repeatedly throughout the experiment and its first run finishes at 900 seconds.

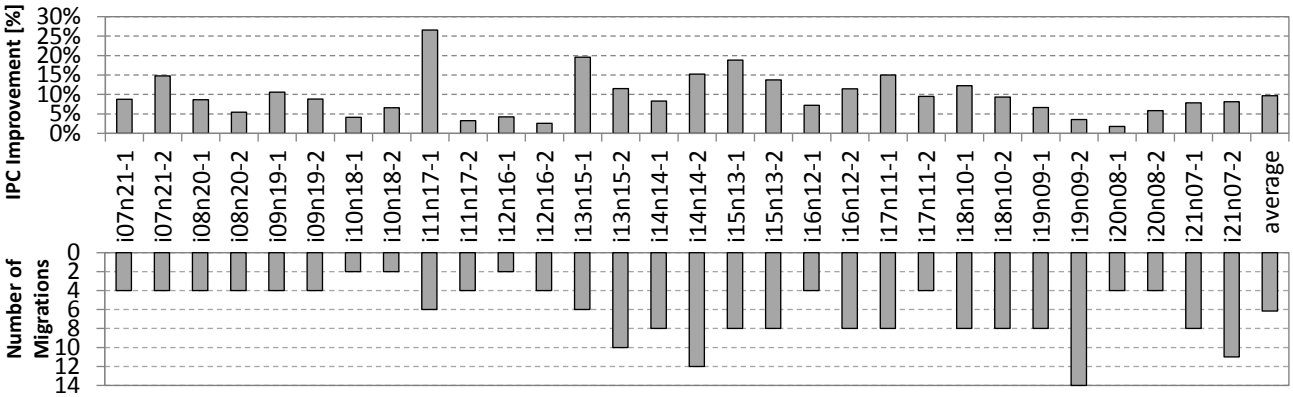


Figure 9: IPC improvement compared to Traditional DRM for different workloads (top), and number of migrations (bottom)

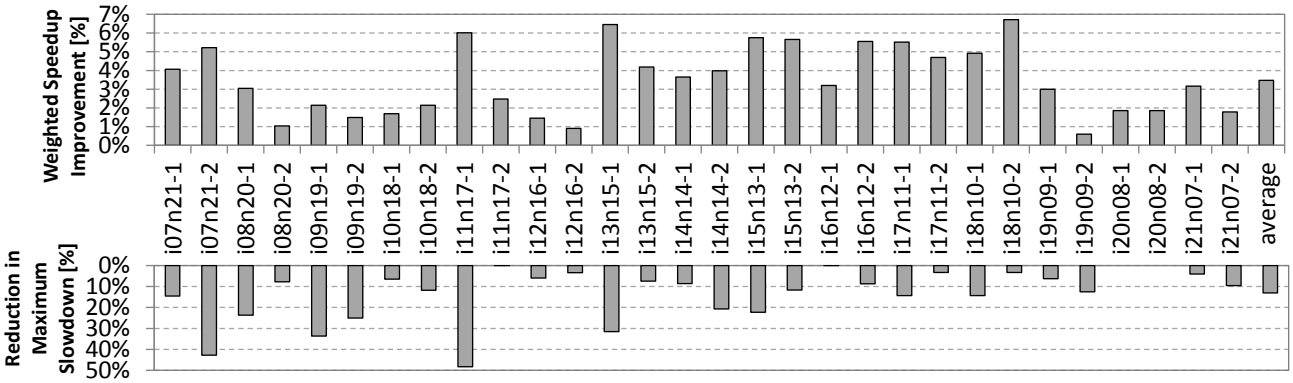


Figure 10: Weighted Speedup (top) and Maximum Slowdown (bottom) normalized to Traditional DRM for different workloads

tant factor to consider for effective virtualized cluster management. Third, *A-DRM* provides the highest improvements when the number of memory-intensive and memory-non-intensive benchmarks is similar, since it is possible to derive significant benefits from balancing demand for cache capacity and memory bandwidth effectively in such workloads.

Figure 9 also shows the number of migrations with *A-DRM*. The number of migrations with traditional DRM is zero, since the CPU/memory capacity utilization among the four hosts are similar, as a result of which no migrations are initiated. On the contrary, for *A-DRM*, this number ranges from 2 to 14 (6 on average).

Figure 10 shows each workload's weighted speedup and maximum slowdown (experienced by any VM in a workload) with *A-DRM* normalized to traditional DRM. *A-DRM*'s average weighted speedup improvement is 3.4% (maximum 6.7%). *A-DRM* reduces the maximum slowdown by 13% (up to 48%). These results indicate that *A-DRM* provides high performance benefits, while also ensuring that VMs are not slowed down unfairly.

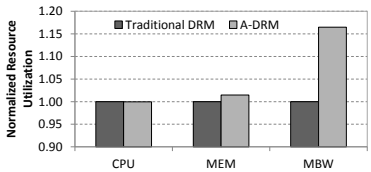


Figure 11: Cluster-wide resource utilization

Figure 11 shows the cluster-wide utilization of CPU, memory capacity and memory bandwidth. We observe that *A-DRM* enhances the memory bandwidth utilization by 17% on average, compared to traditional DRM, while maintaining comparable CPU and memory capacity utilization.

6.4 Parameter Sensitivity

The performance of *A-DRM* can be affected by control knobs such as the $MBW_{Threshold}$, *live migration timeout*, and the sliding window size (see Section 4). To evaluate the impact of these different parameters, we conduct several sensitivity studies with the workload we use in our case study (Figure 7), composed of *STREAM* and *gromacs* (on a two-node cluster).

Memory bandwidth threshold. Figures 12 and 13 show the performance and the number of migrations when we vary the $MBW_{Threshold}$ from 50% to 80%. When $MBW_{Threshold}$ is too small (50%), *A-DRM* identifies a host as bandwidth-starved too often, thereby triggering too many migrations, which can incur high overhead. On the contrary, if $MBW_{Threshold}$ is too large ($> 60\%$), contention might go undetected. In fact, we see that there are no migrations when the $MBW_{Threshold}$ is 70% or 80%. We see that an $MBW_{Threshold}$ of 60% achieves a good tradeoff.

Live migration timeout. Figure 14 shows how the live migration timeout parameter affects *A-DRM*. We vary the parameter from 5 to 60 seconds. When it is too small, the migration essentially defaults to a *stop and copy* technique [35], causing very long downtime. On the other hand, when it is too large, the tracking and transferring of modified pages could consume significant resources, which ag-

gravates the interference problem. We use a live migration timeout of 30 seconds that balances these two factors.

Sliding window size. Figure 15 shows the performance when we set the size of the sliding window to 20, 40, 60 and 80 samples respectively. The sliding window mechanism is applied to get smooth moving average values of the profiled metrics. If the size of the sliding window is too small, the moving average of the profiled data will be sharp, causing spurious changes to trigger *A-DRM* to migrate VMs. On the contrary, if the size of the sliding window is too large, the moving average will change slowly, resulting in *A-DRM* being less adaptive. We see that a value of 80 provides good performance.

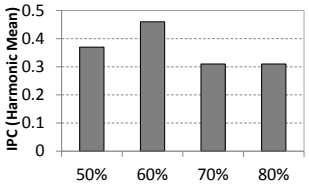


Figure 12: Perf. sensitivity to $MBW_{Threshold}$

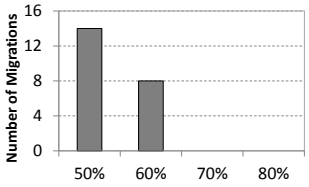


Figure 13: Migrations sensitivity to $MBW_{Threshold}$

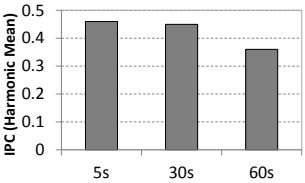


Figure 14: Sensitivity to *live migration timeout*

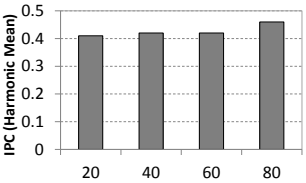


Figure 15: Sensitivity to sliding window size

6.5 Sensitivity to Workload Intensity

Figure 16 shows the IPC improvement when we vary the memory intensity of the workload. Specifically, we use the same benchmarks and the same two-node cluster used in Figure 7, but change the number of VMs that run the memory-*i*ntensive workload. Note that in Figure 7, we only conduct experiments for the *i*07*n*07 case.

This figure shows that when the number of memory-intensive VMs in a workload is less than 5, *A-DRM* hardly performs any migrations, achieving similar performance as traditional DRM. When the number of memory-intensive VMs ranges from 5 to 10, *A-DRM* performs a number of migrations and improves performance significantly. When the number of memory-intensive VMs increases beyond 10, there are very few migrations. Upon further investigation, we discovered the reason for this behavior. Our experimental host has two sockets and each socket has 4-cores, its own on-chip memory controller and local memory (8GB DRAM). Hence, when the number of VMs running memory-intensive applications is smaller than or equal to 4, the socket-level migration (enabled by the NUMA_AUTO_BALANCING feature in the Linux kernel) places each memory-intensive VM on a different socket. Therefore, *A-DRM* does not need to do any migrations. On the contrary, when the number of memory-intensive VMs is larger than 10, all sockets/hosts are similarly congested, which does not offer opportunity for *A-DRM* to kick in and migrate VMs. We conclude that *A-DRM* provides the highest benefits for workloads that have a mix of both memory-intensive and memory-non-intensive applications.

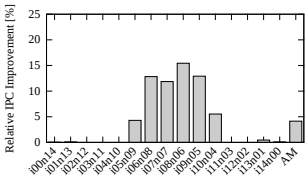


Figure 16: *A-DRM* Performance improvement for different workload intensities

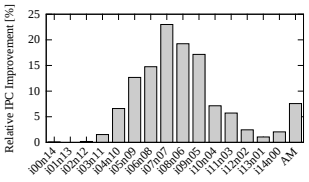


Figure 17: *A-DRM* Performance improvement using socket and host-level migration

6.6 Detecting Per-Socket Contention

While analyzing these results, we were curious to observe what would happen if we turned off the NUMA_AUTO_BALANCING feature and instead, detected contention at a per-socket level and initiate migrations even when a *single socket* on a host is contended (rather than *all* sockets contended). Figure 17 presents results from this experiment.

Comparing Figure 17 with Figure 16 shows that when we detect and respond to socket level interference (while turning off NUMA_AUTO_BALANCING), we achieve better IPC improvement than using just host-level migration. As the number of VMs that run intensive workload increases, the on-chip resources such as LLC and memory controller become greater bottlenecks and socket-level microarchitectural resource contention detection and appropriate VM migration is able to effectively mitigate this interference. We still need to address some issues such as migration cost for socket-level migration and interactions between the hypervisor's NUMA-aware mechanisms and VM migration. We will investigate these issues further as part of our future work.

Another key lesson we learn from this study is that the contention at the inter-socket interconnection network (such as QPI [33] in our system) is not very high. During the initial stages of designing *A-DRM*, we considered the interconnect as one of the contended resources. However, when we measured the QPI traffic, it was always low and hardly ever contended. On analyzing this further, we see that the NUMA_AUTO_BALANCING feature provided by the Linux kernel tries to reduce the remote memory accesses by employing appropriate vCPU scheduling and page migration mechanisms, leading to low interconnect traffic.

7. Related Work

Our work tackles the problem of virtual machine management in a cluster. We have already compared qualitatively and quantitatively to traditional distributed resource management (DRM) schemes (e.g., [31, 34, 72]) that take into account only OS-level metrics such as CPU and memory capacity utilization in making VM-to-host mapping decisions, showing that our proposed *A-DRM* scheme provides significant performance benefits over them.

DRM schemes. To our knowledge, there are few previous works on distributed resource management in a cluster

that are aware of microarchitecture-level interference. One work by Ahn et al. [8] explores this direction by simply sorting all the VMs in a cluster by last level cache misses, and remapping the VMs to minimize the number of LLC misses cluster-wide. Our proposed *A-DRM* scheme is more comprehensive with a detailed cost-benefit analysis, taking into account both memory bandwidth utilization and cache misses.

Various DRM polices have been proposed to manage a virtualized cluster [31, 34, 72]. Wood et al. [72] propose to use CPU, memory and network utilization to detect hot spots and migrate VMs. Isci et al. [34] propose an accurate method to estimate the runtime CPU demand of a workload. Based on this, they design a dynamic resource management policy that consolidates the VMs on under-utilized hosts to other hosts to save energy. VMware DRS [31] is a commercial product that enables automated management of virtualized hosts. DRS constructs a CPU and memory resource pool tree, and estimates the CPU and active memory demand of each VM, each host and each pool. After this estimation phase, a top-down phase is invoked where each child's resource requirement and allocation is checked and maintained. DRS uses an aggressive load balancing policy that minimizes the standard deviation of all hosts' CPU and memory utilization. However, unlike *A-DRM*, none of these schemes are aware of the underlying microarchitecture-level shared resource contention.

Mitigating microarchitecture-level interference through task and data mapping. Several research efforts have developed task and data migration mechanisms within a host, that take into account microarchitecture-level interference [9, 10, 18, 19, 24, 46, 49, 52, 67, 68, 70, 71, 74, 76]. Tang et al. [66] develop an adaptive approach to achieve good thread-to-core mapping in a data center such that threads that interfere less with each other are co-located. Blagodurov et al. [12] observe that contention-aware algorithms designed for UMA systems may hurt performance on NUMA systems. To address this problem, they present new contention management algorithms for NUMA systems. Rao et al. [59, 60] observe that the penalty to access the uncore memory subsystem is an effective metric to predict program performance in NUMA multicore systems. Liu et al. [45] observe the impact of architecture-level resource interference on cloud workload consolidation and incorporate NUMA access overhead into the hypervisor's virtual machine memory allocation and page fault handling routines.

While all these works seek to tune performance within a single node, we focus on a *cluster* of servers in a virtualized environment. In such a virtualized cluster setting, VM migration *across* hosts and DRM schemes that are aware of microarchitecture-level interference enable the ability to mitigate interference that *cannot* be mitigated by migrating VMs within a single host.

Other approaches to mitigate microarchitecture-level interference. Other approaches have been proposed to mitigate microarchitecture-level interference. Some examples of such approaches are interference-aware memory scheduling [39, 41, 42, 48, 50, 51, 63, 64], cache partitioning [36, 40, 57, 65, 73], page coloring [14, 44] and source throttling [13, 20, 38, 54, 55]. These works likely enable more efficient VM consolidation as they enable more efficient and controllable utilization of the memory system, and therefore are complementary to our proposal.

8. Conclusion

We present the design and implementation of *A-DRM*, which, to our knowledge, is the first distributed resource management (DRM) scheme that is aware of and that mitigates microarchitecture-level interference via VM migration. Unlike traditional DRM schemes that operate solely based on operating-system-level metrics, *A-DRM* monitors the microarchitecture-level resource usage (in particular, memory bandwidth and shared last level cache capacity usage) of each virtualized host via an on-chip resource profiler, in addition to operating-system-level metrics like CPU and memory capacity utilization. *A-DRM* then performs a cost-benefit analysis to determine which VMs should be mapped to which hosts and achieves the new mapping through VM migration.

We implement *A-DRM* on a KVM and QEMU platform. Our extensive evaluations show that *A-DRM* can enhance the performance of virtual machines by up to 26.55% (average of 9.67%), under various microarchitecture interference levels, compared to a traditional DRM scheme that is not aware of microarchitecture-level interference [31, 34, 72]. *A-DRM* also improves the average cluster-wide memory bandwidth utilization by 17% (up to 36%).

Our work demonstrates a promising path to achieve substantial improvements through microarchitectural-interference-aware distributed resource management for virtualized clusters. Our results show that being aware of microarchitecture-level shared resource usage can enable our *A-DRM* scheme to make more effective migration decisions, thus improving performance and microarchitecture-level resource utilization significantly. We propose to explore more sources of microarchitectural interference (e.g., interconnect bandwidth [16–18, 25, 26]) to further improve performance and resource utilization in virtualized clusters, as part of future work.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We acknowledge the support of our industrial partners, including Facebook, Google, IBM, Intel, Microsoft, Qualcomm, Samsung, and VMware. This work is partially supported by NSF (awards 0953246, 1212962, 1065112), the Intel Science and Technology Center on Cloud Computing, the National High-tech R&D Program of China (863) under grant No. 2012AA01A302, National Natural Science Foundation of China (NSFC) under Grant Nos. 61133004, 61202425, 61361126011. Hui Wang is partially supported by the China Scholarship Council (CSC). Lavanya Subramanian is partially supported by a Bertucci fellowship.

References

[1] Windows Azure. http://www.windowsazure.com/en-un/.

[2] Amazon EC2. http://aws.amazon.com/ec2/.

[3] libvirt: The virtualization API. http://libvirt.org.

[4] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[5] QEMU. http://qemu.org.

[6] SPEC CPU2006. http://www.spec.org/spec2006.

[7] STREAM Benchmark. http://www.streambench.org/.

[8] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *HotCloud*, 2012.

[9] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT*, 2010.

[10] N. Beckmann, P.-A. Tsai, and D. Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. In *HPCA*, 2015.

[11] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *USENIX ATC*, 2011.

[13] K. K. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu. HAT: heterogeneous adaptive throttling for on-chip networks. In *SBAC-PAD*, 2012.

[14] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO*, 2006.

[15] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[16] R. Das, O. Mutlu, T. Moscibroda, and C. Das. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*, 2009.

[17] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das. Aérgia: exploiting packet latency slack in on-chip networks. In *ISCA*, 2010.

[18] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *HPCA*, 2013.

[19] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *ASPLOS*, 2013.

[20] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS*, 2010.

[21] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Bandwidth Bandit: Quantitative characterization of memory contention. In *PACT*, 2012.

[22] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, (3), 2008.

[23] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, 2008.

[24] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SoCC*, 2011.

[25] B. Grot, S. W. Keckler, and O. Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *MICRO*, 2009.

[26] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *ISCA*, 2011.

[27] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[28] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO load balancing across storage devices. In *FAST*, 2010.

[29] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, 2010.

[30] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online storage performance management in virtualized datacenters. In *SoCC*, 2011.

[31] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, and X. Zhu. VMware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[32] Intel. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors.

[33] Intel. An Introduction to the Intel QuickPath Interconnect, 2009.

[34] C. Isci, J. Hanson, I. Whalley, M. Steinder, and J. Kephart. Runtime demand estimation for effective dynamic resource management. In *NOMS*, 2010.

[35] C. Isci, J. Liu, B. Abali, J. Kephart, and J. Kouloheris. Improving server utilization using fast virtual machine migration. *IBM Journal of Research and Development*, 55 (6), Nov 2011.

[36] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.

[37] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *SC*, 2012.

[38] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. Managing GPU concurrency in heterogeneous architectures. In *MICRO*, 2014.

[39] H. Kim, D. de Niz, B. Andersson, M. H. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[40] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[41] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[42] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[43] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, 2007.

[44] 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 *HPCA*, 2008.

[45] M. Liu and T. Li. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *ISCA*, 2014.

[46] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[47] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[48] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.

[49] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *MICRO*, 2011.

[50] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.

[51] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.

[52] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *EuroSys*, 2010.

[53] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX ATC*, 2005.

[54] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next generation on-chip networks: What kind of congestion control do we need? In *HotNets*, 2010.

[55] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects. In *SIGCOMM*, 2012.

[56] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, 2009.

[57] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[58] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[59] J. Rao and X. Zhou. Towards fair and efficient SMP virtual machine scheduling. In *PPoPP*, 2014.

[60] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *HPCA*, 2013.

[61] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *PACT*, 2012.

[62] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[63] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA*, 2013.

[64] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu. The blacklisting memory scheduler: Achieving high performance and fairness at low cost. In *ICCD*, 2014.

[65] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1), 2004.

[66] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[67] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for QoS in warehouse scale computers. In *CGO*, 2012.

[68] A. Tumanov, J. Wise, O. Mutlu, and G. R. Ganger. Asymmetry-aware execution placement on manycore chips. In *SFMA*, 2013.

[69] H. Vandierendonck and A. Seznec. Fairness metrics for multithreaded processors. *IEEE CAL*, February 2011.

[70] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[71] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *HPDC*, 2011.

[72] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[73] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.

[74] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *ISCA*, 2013.

[75] K. Ye, Z. Wu, C. Wang, B. Zhou, W. Si, X. Jiang, and A. Zomaya. Profiling-based workload consolidation and migration in virtualized data centres. *TPDS*, 2014.

[76] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.