iBench: Quantifying Interference for Datacenter Applications

Christina Delimitrou and Christos Kozyrakis
Electrical Engineering Department
Stanford University
{cdel, kozyraki}@stanford.edu

***Abstract*—Interference between co-scheduled applications is one of the major reasons that causes modern datacenters (DCs) to operate at low utilization. DC operators traditionally sidestep interference either by disallowing colocation altogether and providing isolated server instances, or by requiring the users to express resource reservations, which are often exaggerated to counter-balance the unpredictability in the quality of allocated resources. Understanding, reducing and managing interference can significantly impact the manner in which these large-scale systems operate.**

We present iBench, a novel workload suite that helps quantify the pressure different applications put in various shared resources, and similarly the pressure they can tolerate in these resources. iBench consists of a set of carefully-crafted benchmarks that induce interference of increasing intensity in resources that span the CPU, cache hierarchy, memory, storage and networking subsystems. We first validate the effect that iBench workloads have on performance against a wide spectrum of DC applications. Then, we use iBench to demonstrate the importance of considering interference in a set of challenging problems that range from DC scheduling and server provisioning, to resource-efficient application development and scheduling for heterogeneous CMPs. In all cases quantifying interference with iBench results in significant performance and/or efficiency improvements. We plan to release iBench under a free software license.

I. Introduction

An increasing amount of computing is performed in the cloud, primarily due to the flexibility and cost benefits both for the end-users and the operators of these large-scale datacenters (DCs). Public cloud providers such as Amazon EC2 [1], Microsoft Windows Azure [43] and Google Compute Engine [21] host tens of thousands of applications on a daily basis. Additionally, several companies organize their IT in private clouds, using management systems such as VMWare vCloud [42], Eucalyptus [19], Citrix XenServer [46] or OpenStack [33].

Ideally, these systems should provide scalable compute at low cost (*resource efficiency*). In the past, cost efficiency has improved by switching to commodity servers [7] and reducing the cost of the power delivery and cooling infrastructure [23]. Similarly DC compute capabilities have scaled by increasing the number of datacenters, adding more servers per infrastructure and relying on the chip technology to provide increased compute for the same power budget. However, these techniques are reaching the point of diminishing returns. Recent power usage effectiveness (PUE) numbers show that the overhead for power delivery and cooling is less than 10% [6], [32]. Similarly, building more DCs requires significant investment while chip technology starts to progress at a slower pace, as we are reaching the end of voltage scaling [18]. To further improve DC operation, we must focus on *using efficiently* the tens of thousands of servers available in each large-scale infrastructure.

There are, however, several challenges towards achieving this goal. Server utilization in datacenters today is notoriously low [6], [7] for reasons that include load changes, difficulty in provisioning servers appropriately [27], widely varying workload characteristics and constraints, and platform heterogeneity, which occurs as servers get gradually deployed and replaced over the typical 15-year provisioned lifetime of a DC [6], [7], [23]. More importantly, increasing utilization involves co-scheduling applications in the same machine, which results in performance degradation due to *interference* in shared resources [17], [28]. DC operators typically sidestep interference issues either by disallowing colocation or by resorting to exaggerated resource reservations to counterbalance performance unpredictability. A driver behind the inefficiency of these approaches is the limited visibility both users and DC operators have into workload characteristics. Fig. 1 for example, shows the memory capacity and memory bandwidth requirements of a wide set of application types, including single-threaded (ST) and multi-threaded (MT) benchmark suites such as SPECCPU2006, PARSEC [10], SPLASH-2 [45], BioParallel [24] and MineBench [31], multi-programmed (MP) mixes of these workloads, distributed batch (Hadoop) and latency-critical (memcached) applications, as well as traditional relational database workloads (MySQL). Capacity and bandwidth demands are normalized to the provisioned system values. The size of each bubble corresponds to the size of each job (number of tasks or clients). It becomes obvious that even when looking only at memory requirements, demands vary widely. Therefore, understanding the sensitivity workloads have to contention is critical towards reducing and managing interference in a way that enables QoS-aware operation at high utilization.

Previous work has shown the importance of accounting for interference in DC scheduling [17], [28] and has developed hardware and software mechanisms to minimize interference effects. Mars et al. [28] show that ignoring the interference chracteristics of large DC applications in the memory subsystem can cause significant performance degradations that violate the workloads' QoS constraints. Typically, determining the interference profile of a workload involves either retroactively observing which co-scheduled applications contend in shared resources and annotating the offending workloads [47] or profiling the workload against a carefully-crafted benchmark that puts pressure on a specific shared resource [17], [28]. The

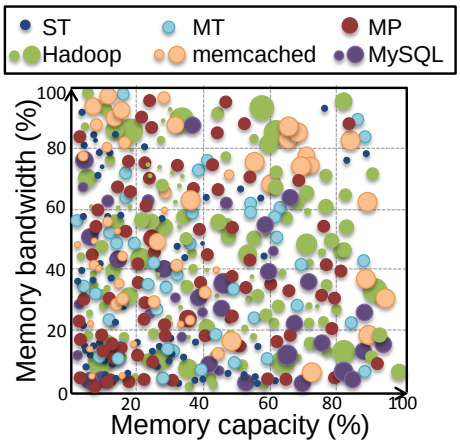

Fig. 1: Pressure in memory capacity and memory bandwidth from a wide set of applications, as measured by iBench. The bubble size is proportional to the number of tasks (for Hadoop) or clients (for memcached) of the corresponding application.

disadvantage of the first approach is that interference is determined after performance degradation has occured, and, currently, requires manual annotation of contending workloads. The second approach is less invasive, enables interference detection before this reflects into performance degradation, but requires effort in designing targeted benchmarks that put pressure on specific resources. Currently, there is no open-source benchmark suite that enables fast characterization of the interference an application tolerates and causes in various subsystems.

In this paper we present iBench, a novel benchmark suite that helps quantify the sensitivity of DC (and conventional) applications to interference. iBench consists of a set of carefully-crafted benchmarks that generate contention of tunable intensity in various shared resources which include the core, the cache and memory hierarchy, and the storage and networking subsystems. iBench workloads are called SoIs (sources of interference). Injecting an SoI in a machine hosting an application identifies the interference that application can tolerate in the corresponding shared resource before it violates its QoS, and the interference it itself creates. We validate iBench against a set of DC applications that range from distributed frameworks such as Hadoop [2], latency-critical online services like memcached [30] and conventional single-threaded, multithreaded and multiprogrammed single-node applications, and verify the accuracy and consistency of the interference measurements.

We have used iBench in various system studies, and specifically in this work we show that it improves decision quality in four use cases that extend to DC and CMP systems and span hardware and software challenges. First, we use the benchmark suite to quantify the interference sensitivity of a large set of applications resembling a cloud provider mix and use this information to make resource-efficient scheduling decisions. Second, we use iBench to guide the hardware configuration of DC servers, such that the system is appropriately provisioned to tolerate the pressure workloads put in different resources. Third, we move the interference characterization one step in advance and use it to guide application software development, before the workload's full deployment. iBench here is used to determine the resources where an application induces contention, and to assist the software developer to design more resource-efficient code. Given the speed of interference characterization, using iBench significantly accelerates the iterative testing process of application software. Finally, we show that iBench is applicable to studies outside datacenters and use the interference characterization to guide scheduling decisions in a large-scale heterogeneous CMP. Note that in this case characterization needs to also account for the different core designs, while being lightweight and trasparent to the workload. In all cases, using iBench significantly improves the system's ability to preserve QoS guarantees in a resource-efficient manner. Specifically, scheduling in a DC using iBench preserves performance for the majority of workloads, while significantly increasing utilization, by 42%. Also, by revising code regions, based on indications from iBench, we managed to reduce the application footprint of a large, data mining application by 49%, *while* speeding up the workload by 35%. We plan to release iBench under a free software license.

The rest of this paper is structured as follows. Section II discusses related work. Section III describes the iBench workloads, while Section IV includes a validation of interference measurements using iBench. Section V presents the four system studies used to evaluate iBench. Finally, Section VI presents topics for future work and concludes the paper.

II. Related Work

DC benchmark suites: A major roadblock when studying DC applications is the unavailability of representative workloads and input loads. Given this challenge, there is extensive work on characterization and modeling of DC applications [5], [14], [15], [16], [25], [38] that leads to generated workloads with characteristics that closely resemble those of the original application. The generated workloads can then be used in system studies without the limitation of needing access to real DC workloads. While this is a viable approach in some cases, modeling has limitations; there are workload aspects that are not captured in the model to preserve simplicity. However omitting these aspects can cause the generated workload to deviate from its expected behavior. Additionally, modeling is more applicable to large, long-running applications that can be characterized in detail to provide some input to the model, but is less beneficial in systems like Amazon's EC2 or Windows Azure where submitted workloads are typically unknown and no a priori assumptions can be made about their behavior.

A different track to side-step DC workload unavailability is the design of open-source versions of popular applications, that resemble their behavior and structure. Examples of such workloads are Lucene [3] and Nutch [4] for Websearch, Roundcube [35] for Webmail, or Hadoop [2] for MapReduce [13]. In the same spirit, CloudSuite [20] is an open-source benchmark suite that aggregates a set of such applications, including

data analytics, media streaming and web serving. While open-source applications cannot be exact replicas of production-class workloads, they provide a reasonable approximation of their behavior.

Interference-related workloads: Recent work has shown that reducing interference is critical to preserving application performance in DCs [17], [22], [28], [40]. Govindan et al. [22] designed a synthetic cache loader to profile an application's cache behavior and the pressure it would put on co-scheduled workloads. Similarly, to demonstrate the impact of interference in the memory subsystem, Mars et al. [28] designed two micro-kernels that create tunable contention in memory capacity and memory bandwidth. These kernels are then used to quantify the sensitivity of a workload to memory interference. Additionally, Tang et al [40] designed SmashBench, a benchmark suite for cache and memory contention. Benchmarks include operations on binary search trees (BSTs), arrays and 3D arrays.

With iBench we extend the resources in which interference is quantified to the core, the memory hierarchy, and the storage and networking subsystems. This enables iBench to provide critical insights on the sensitivity of applications to resource contention that can guide both software (e.g., scheduling) and hardware (e.g., server provisioning) system studies.

III. iBench Workloads

A. Overview

The goal of iBench is to identify the shared resources an application creates contention to, and similarly the type and amount of contention the application is sensitive to. For this purpose, all iBench workloads have tunable intensity that progressively puts more pressure on a specific shared resource until the behavior of the application changes (i.e., performance degrades). A similar technique has been shown to provide accurate estimations on sensitivity to contention in the memory subsystem [28], [40]. In total, iBench consists of 15 carefully-crafted workloads, which we call sources of interference (SoIs), each for a different shared resource. The following section describes each one of them in detail. To provide some proportionality between the intensity of the benchmark and its impact to the corresponding resource, SoIs are designed such that their impact increases almost linearly with the intensity of the benchmark. Finally, we try to ensure that the impact of the different iBench workloads is not overlapping, e.g., that the memory bandwidth SoI does not cause significant contention in memory capacity and vice versa. Section IV validates that this is indeed the case across SoIs.

B. Designing the SoIs

Memory capacity (SoI1): This kernel progressively accesses larger memory footprints until it takes over the entire memory capacity. The access pattern of addresses in this case is random, but can also be set to perform strided memory accesses. The following snippet shows the basic operation of SoI1:

```
t = 0;
while (t < duration) {
     ts = time(NULL);
     while (coverage < x%) {
          // SSA: to increase ILP
          access[0] += data[r] << 1;
          access[1] += data[r] << 1;
           ...
          access[30] += data[r] << 1;
          access[31] += data[r] << 1;
          wait(t_x/acc_x);
      }
      x++;
      t += time(NULL) - ts;
 }
```

The kernel identifies automatically the size of memory available in the system and scales its footprint "almost" proportionately with time. From the snippet above, t is the total time the SoI will run for. The benchmark uses single static assignment (SSA) to increase the ILP in memory accesses, and launches as many requests as necessary to guarantee the appropriate capacity coverage at each point during its execution, e.g., at 8% intensity, capacity coverage should be 8%. The memory addresses r are selected randomly with a low-overhead random generator function. For low intensities the kernel may switch to an idle state between memory requests. t_x is the time the kernel spends at a specific intensity level, and is a function of the benchmark duration t and the intensity level x. acc_x is the number of accesses required to reach a specific coverage level and is also a function of the intensity x. The time the kernel can remain idle is proportional to t_x and inversely proportional to acc_x. As the kernel moves to higher intensities, the fraction of time the kernel remain idle reduces as more accesses are required to achieve a certain memory coverage. By default all kernels run for 10msec, however duration is a configurable parameter.

Memory bandwidth (SoI2): The benchmark in this case performs streaming (serial) memory accesses of increasing intensity to a small fraction of the address space. The intensity increases until the SoI consumes 100% of the sustained memory bandwidth of the specific machine. The intensity of accesses increases linearly with the memory bandwidth used. The reason why accesses happen to a relatively small fraction of memory (e.g., 10%) is to decouple the effects of contention in memory bandwidth from contention in memory capacity. The following snippet captures the main operation of the streaming kernel:

```
t = 0;
while (t < duration) {
   ts = time(NULL);
   for (int cnt = 0; cnt < acc_x; cnt++) {
          access[cnt] = data[cnt]*data[cnt+4];
          wait(t_x/acc_x);
    }
```

```
    x++;
    calculate acc_x;
    t += time(NULL) - ts;
}
```

The definition of t_x and acc_x is the same as before. In the subsequent SoIs we skip the code snippets in the interest of space, and describe their main operation.

Storage capacity (SoI3): Storage corresponds to the non-volatile secondary devices, e.g., disk drives or flash that store data. We assume these are disk drives for simplicity. The microbenchmark accesses random data segments across the disk's sectors. The amount of accessed data increases linearly with the SoI's intensity, i.e., at 20% intensity close to 20% of disk capacity is accessed by the SoI.

Storage bandwidth (SoI4): This benchmark creates traffic of increasing intensity to the hard drives of the system. Disk accesses in this case are serial and the consumed disk bandwidth increases almost linearly with the intensity of the SoI, e.g., at 100% intensity, the SoI uses close to 100% of the sustained disk bandwidth of the system.

Network bandwidth (SoI5): This SoI is of interest to workloads with network connectivity, e.g., online services or distributed frameworks like MapReduce. It operates by issuing network requests of increasing intensity (size and frequency of requests) to a remote host. We currently do not deploy rate limiting mechanisms, therefore the SoI can take over 100% of the available network bandwidth, essentially starving any co-scheduled application.

Last level cache (LLC) capacity (SoI6): The benchmark mines the `/proc/cpuinfo` of the system and adjusts its footprint, access pattern and the pace that its intensity increases based on the size and associativity of the specific LLC. The kernel issues random accesses that cover an increasing size of the LLC capacity. Because caches are structured in sets, it is easy to mathematically prove and practically guarantee that the footprint of the benchmark increases linearly with the intensity of the SoI and that its accesses are uniformly distributed. We skip the proof in the interest of space. Finally, to guarantee that accesses are not intercepted in the lower levels of the hierarchy (L1, L2) we concurrently run small tests that sweep the smaller caches (without introducing additional misses) to ensure that all accesses from the SoI go to the LLC.

LLC bandwidth (SoI7): This benchmark is similar to the SoI for memory bandwidth in that it performs streaming data accesses to the LLC. In this case the size and peak bandwidth the SoI targets are tuned to the parameters of the specific last level cache. Because accesses are streaming over a fraction of the cache, the lower levels of the hierarchy do not play as important a role as with random accesses. We have found that running the sweep tests for L1 and L2 does not make a significant difference when measuring sensitivity to contention in LLC bandwidth.

L2 capacity (SoI6'): This is a similar benchmark to SoI6 (LLC Capacity), and is applicable in systems with 3+ levels of cache hierarchy. The footprint in this case grows up to the L2 cache size and the L2 associativity is used to tune how intensity changes over the kernel's duration.

L2 bandwidth (SoI7'): Similar to SoI7 (LLC bandwidth), but tuned to the size and associativity of the L2. Accesses in this case are streaming.

L1 i-cache (SoI8): A simple kernel that sweeps through increasing fractions of the i-cache, until it populates its full capacity. Accesses in this case are again random.

L1 d-cache (SoI9): A copy of the previous SoI, tuned to the specific structure and size of the d-cache (typically the same as the i-cache).

Translation lookahead buffer (TLB) (SoI10): This benchmark fetches pages from memory at increasing rates until it occupies all the TLB entries. This forces long page walks for any co-scheduled application, inducing high performance degradations. Again, because of the structure of TLBs it is easy to compute the pace at which SoI intensity should increase to guarantee a linear relation with the occupied entries.

Integer processing units (SoI11): While the core can be approached as a single shared resource, we decide to separate the different types of operations to integer, floating point (and an optional vector SoI when SSE extensions are available). All three SoIs are assembly-level benchmarks that issue an increasing number of the corresponding type of instructions. For SoI11 these are instructions between integers.

FP processing units (SoI12): Similarly here, floating point instructions are issued at an increasing rate. SoIs 11 and 12 (and 15 when applicable) can run both on the same hardware thread and on different threads sharing the same core.

Prefetchers (SoI13): This benchmark tries to inject unpredictability in the instructions the prefetcher brings from memory, and decrease its effectiveness. This may seem similar to the operation of the L1 i-cache benchmark, however the prefetcher SoI employs a different access pattern than SoI8. Instead of simply sweeping through the L1 and evicting the co-runner's instructions, the SoI here is a small program that only takes up a fraction of the L1 i-cache, but interleaves its instructions with the examined application's instructions. This way the prefetcher gets tricked into bringing the SoI's next "expected" instructions from memory instead of the primary application's. Intensity here translates to the time the SoI is non-idle. This SoI also interacts in part with the system's branch predictor.

Interconnection network (SoI14): This benchmark is designed using message passing primitives between cores. As the SoI intensity goes up the number and fanout of messages sent by the kernel increases. For high intensities the injected traffic becomes adversarial, leading the remaining system cores to starvation.

Vector processing units (SoI15): This SoI is only applicable in systems with SIMD ISA extensions, e.g., SSE3/4. It takes advantage of these extensions to launch 256-wide SIMD instructions with increasing frequency. Instructions are issued

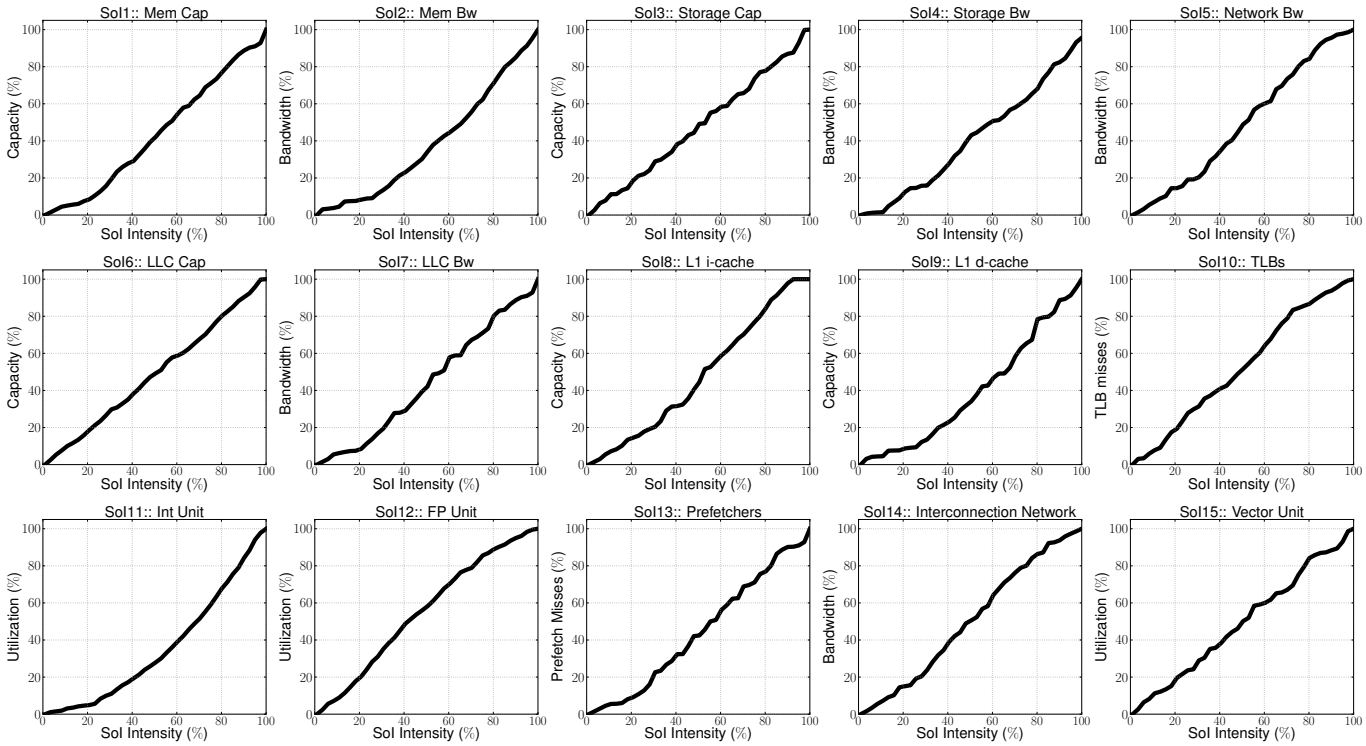

Fig. 2: The iBench workloads. For each benchmark we show the system impact for increasing SoI intensity. We do not include the graphs for the L2 capacity and bandwidth SoIs. These are similar to the ones for LLC capacity and bandwidth.

on a small data set to avoid interfering with the cache/memory hierarchy. None of the systems we tested uses extensive memoization techniques therefore operating on the same data does not reduce the load to the vector units.

IV. Validation

We want to validate three aspects of iBench; first that the benchmarks indeed induce contention in their corresponding resources, and that their impact increases almost linearly with their intensity. Second, we want to evaluate the impact of iBench on conventional and DC applications and verify that the SoIs can be used to detect sensitivity to interference. Finally, we want to verify that the different SoIs do not overlap with each other in a way that voids the insights drawn about an application's behavior, e.g., that the memory bandwidth SoI does not introduce significant contention in memory capacity.

A. Individual SoIs Validation

Fig. 2 shows the impact of the 15 SoIs across their intensity spectrum (0-100%). For capacity-related benchmarks we show their cache, memory or disk footprint. For the bandwidth-related SoIs we show the fraction of bandwidth they consume normalized to the provisioned sustained cache, memory or disk bandwidth. For the core-related SoIs we show the utilization they induce in the corresponding functional units (int, fp or vector). Finally, for the TLB benchmark we show TLB misses and for the prefetcher benchmark, prefetch misses. All measurements are collected using performance counters on a dual-socket, 8-core Nehalem server with private L1s and L2s and a shared 8MB L3 cache and 32GB of RAM. The server has a 1GB NIC and 4 500GB hard drives. Each SoI runs for 10msec on its own and covers its full range of 0 to 100% of intensity. Each SoI automatically detects the system parameters that it needs in order to adjust its operation, e.g., cache or TLB size, core count or NIC type. From Fig. 2 we see that for all benchmarks the impact to the corresponding resource increases almost linearly with their intensity. The only SoIs that slightly deviate from linear are the core-related benchmarks Int and FP. This happens because correlating the number of issued instructions to the eventual system utilization is harder than correlating the number of cache accesses to the capacity used. We plan to further refine these workloads to better approach linear load increase as part of future work.

B. SoI Impact on Applications

iBench is aimed to detect and quantify the sensitivity of DC and conventional workloads to various sources of interference. Here we validate that this operation is accurate. We inject iBench workloads in a conventional application (mcf from the SPECCPU2006 suite) and in a DC latency-critical application (memcached [30]) and measure their sensitivity to interference in the corresponding resources. Each application runs on a single server, and memcached is set up with 1000 clients launching 40,000 QPS in total, with a target per-request latency of 200usec. mcf is profiled for 10msec against the LLC capacity SoI, and memcached against the network bandwidth SoI. Both SoIs inflate to full intensity (100%). Fig. 3a, b shows the results for mcf and Fig. 3c, d for memcached.

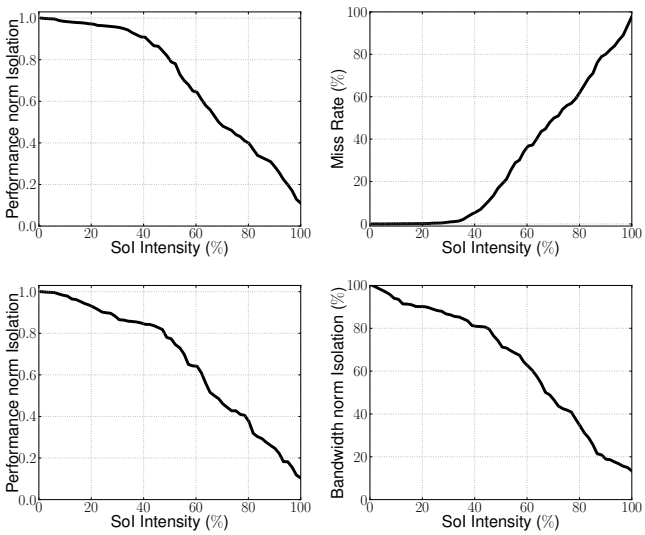

Fig. 3: Validation of the impact contention generated using iBench has on mcf and memcached. Fig. 3a shows the performance of mcf when co-scheduled with the LLC capacity SoI, while Fig. 3b shows its new miss rate curve as SoI intensity increases. Fig. 3c shows the performance of memcached when running with the network bandwidth SoI and Fig. 3d shows its bandwidth share compared to when running alone.

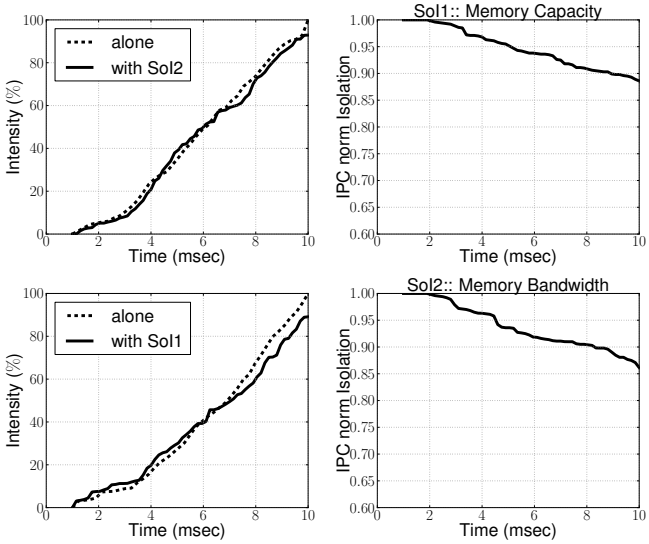

Fig. 4: Validation of the impact SoIs have on each other. Fig. 4a shows the intensity of SoI1 when co-scheduled with SoI2 and similarly for SoI2 (Fig. 4c). Fig. 4b, d show the achieved IPC normalized to target when the two SoIs run together. Overall, interference between benchmarks is minimal.

Fig. 3a shows the performance impact of contention in LLC capacity for mcf and the corresponding miss rate curve as the intensity of the SoI increases. Comparing the two shows that the SoI indeed induces significant performance degradation to the application due to cache contention. The point when performance gets a significant hit coincides with the moment when the miss rate increases rapidly, therefore the SoI is correctly stressing its target resource. Similarly, for memcached we show the performance impact from increased contention in the network and the bandwidth fraction memcached manages to extract compared to the target fraction it needs to preserve its performance requirements. Again there is a direct correlation between performance degradation and its cause. As SoI intensity increases, the goodput of memcached (fraction of requests that meet their target latency) rapidly decreases. Fig. 3d shows the reason behind this degradation. For high SoI intensities, the bandwidth share of memcached becomes increasingly smaller, introducing queueing delays to incoming requests. At the same time, examining its cache miss rate or memory behavior does not show significant variations compared to when memcached is running alone. This verifies that the SoI is confined to its specific resource and does not violently disrupt the utilization of other subsystems. We further validate this observation in the following subsection. We have also verified that these results are consistent across the different SoIs for various workload types.

C. Correlation between SoIs

Finally, we verify that different sources of interference (SoIs) do not overlap and interfere with each other. For this purpose we co-schedule two SoIs at a time in the same core of the 8-core system previously used. Fig. 4 shows the increase in intensity and corresponding performance normalized to isolation for a co-schedule of the memory capacity and memory bandwidth SoIs. As shown in the figure, for high system loads, there is a small impact in the ability of each SoI to reach its full intensity. Similarly there is a slight degradation in performance compared to running in isolation. However, for both SoIs degradations are mild, which means that the different benchmarks do not induce significant contention outside their target resource. This is important to both obtain accurate interference measurements, and make valid assumptions on their causes. We have performed this experiment with different SoI combinations with similar results.

V. Use Cases

A. Datacenter Scheduling

Currently, DC operators often disallow application co-scheduling in shared servers to preserve QoS guarantees. However, this leads to serious resource underutilization. On the other hand, co-scheduling applications can induce interference due to contention in shared resources. We use iBench to quantify the tolerance a workload has to various sources of interference, and similarly the interference it causes in shared resources. Given this information, a scheduler determines the applications that can be safely co-scheduled without performance degradation from interference. For this use case, the scheduler simply tries to minimize:

$$||i_t - i_c||_{L_1} \quad (1)$$

where i_t and i_c the tolerated and caused interference for two examined applications. The tolerated interference is calculated as described in Section IV. The caused interference is similarly

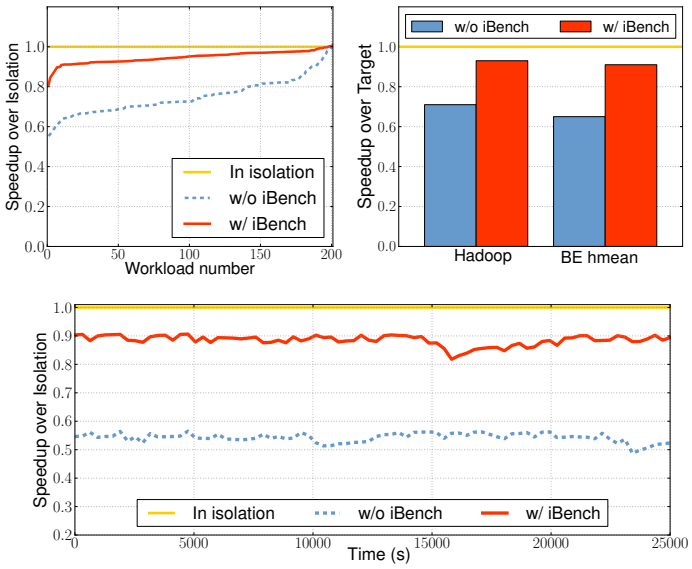

Fig. 5: Performance achieved by a scheduler that accounts for interference using iBench compared to a system that ignores interference in scheduling decisions. Results are shown for three scenarios: a cloud workload mix (Fig. 5a), a distributed workload (Fig. 5b) and a latency-critical application (Fig. 5c).

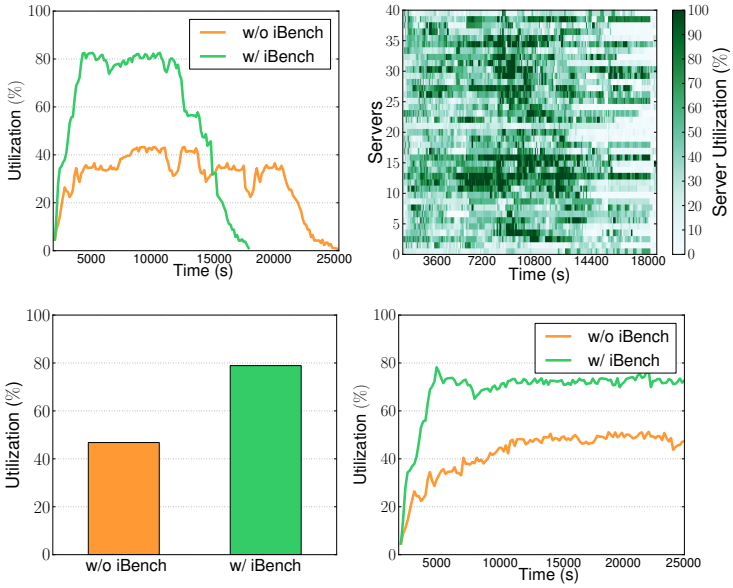

Fig. 6: Utilization achieved by a scheduler that uses iBench compared to a system that ignores interference in scheduling decisions. For the first scenario fewer machines are needed (and for less time) (Fig. 6a, b), while for the second (Fig. 6c) and third scenarios (Fig. 6d) more best-effort applications are co-scheduled with the primary workload.

calculated, by quantifying the impact the examined workload has on the performance of an SoI. The $L1$ norm is calculated across the different SoIs. More sophisticated scheduling techniques can be deployed to take better advantage of the information provided by iBench [17]. Obviously applications can change behavior during their execution. This is especially true for DC workloads [7], [29]. The scheduler adapts to these changes to preserve QoS throughout an application's execution. If at any point in time it detects that an application is running under its QoS, the scheduler injects iBench workloads to the system to construct a new interference profile. Any further scheduling decisions use the new interference profile. Required migrations due to behavior changes are handled by a low-overhead live migration system present in the cluster. In the event where migration is not possible, the scheduler disallows additional applications to be placed on the same machine as the affected workload.

We design three scenarios; first a cloud workload mix that resembles a system like EC2, where 200 applications are submitted in a 40-machine cluster with 1 sec inter-arrival times. All nodes are dual socket, 4-12 core machines with private L1 and L2 caches and shared L3 caches, and 16-48GB of RAM. All applications are selected randomly from a pool consisting of the full SPECCPU2006 suite, 22 workloads from PARSEC [10], SPLASH-2 [45], BioParallel [24] and Minebench [31], 140 multiprogrammed workloads of 4 SPEC applications each, based on the methodology in [37], and 10 I/O-bound data mining workloads [34]. The second scenario involves a Hadoop workload running distributed on 40 nodes with low-priority best-effort (BE) applications occupying the remaining server capacity, and the third scenario involves a 40-node installation of *memcached*, running as the primary process and best-effort applications using the remaining resources. Fig. 5a compares application performance for the first scenario when quantifying interference using iBench, against a baseline scheduler that only considers the CPU and memory requirements of an application and assigns workloads to least-loaded (LL) servers (*w/o iBench*). The latter is common practice in many cloud providers today [41]. Performance is normalized to running in isolation and applications are ordered from worst- to best-performing. Using iBench to quantify the pressure applications put on various system resources improves performance, by 15.7% on average and up to 25%. Similarly, performance improves in the second and third scenario both for the primary workloads (Hadoop and memcached respectively) and the best-effort applications. Managing interference is beneficial to utilization as well, since more applications can be scheduled on the same machine. Fig. 6 shows the utilization for each of the three scenarios when accounting for interference using iBench and when using the baseline least-loaded (LL) scheduler. The benefits are twofold; first utilization increases, improving resource-efficiency for the DC operator (Fig. 6a). Second, the duration of the scenario reduces because applications are running near their target performance. Fig. 6b offers a closer look at utilization across the different servers in the cluster throughout the scenario's execution. The increase in utilization is also consistent for the other two scenarios (35.6% and 27.1% respectively on average). There is still some performance degradation in these scenarios, which iBench cannot prevent. This is due to

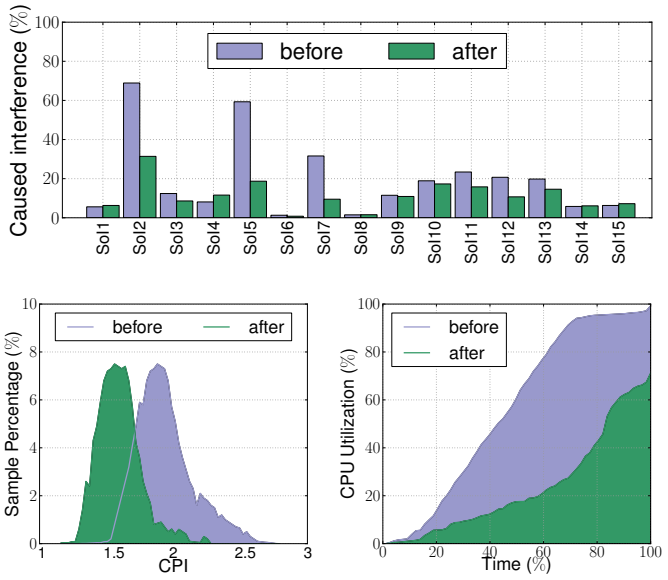

Fig. 7: Using iBench to provision a server that hosts a specific workload improves performance and reduces resource contention. Fig. 7a compares the old and new interference profiles of the workload. Fig. 7b shows the CPI distribution for memcached in the original system configuration and after reconfiguring the system based on the interference profile from iBench, and Fig. 7c shows that utilization decreases, as resources are appropriately balanced to reduce contention.

fast-changing workloads, complex applications that introduce inaccuracies in interference measurements, or workloads that have pathologies when co-scheduled with specific applications. Additional mechanisms can be used to address these issues.

B. Server Provisioning

Provisioning servers is especially difficult for cloud providers that have to accommodate any - possibly unknown - submitted workload. Even in the case of well-studied, long-running applications the datacenter architect must deal with evolving application code and varying user patterns. Here we use the output of iBench to guide the way system resources are balanced in a DC server running memcached [30]. The workload runs with 1000 clients launching a total of 40,000QPS with a latency constraint of 200usec. Fig. 7a shows the interference profile of the application running on a default server configuration (4 cores, 8MB L3, 16GB RAM, 1GB NIC) across the different SoIs. It is evident that the application puts significant pressure on the cache hierarchy and the network and memory subsystems (SoI2: memory bandwidth, SoI5: network bandwidth, SoI7: LLC bandwidth and SoI11-12: core). Based on this information we adjust the parameters of the system. To alleviate the contention in the memory hierachy we switch to a triple-memory channel server with 24GB of total memory capacity. Similarly, we move from a server with a 1GB to a 10GB NIC to accommodate the application's network demands. We maintain the core count and the rest of the system parameters the same. Fig. 7a also shows the new interference profile, where both the contention in the cache/memory hierarchy and the network subsystem are now significantly reduced. Fig. 7b shows the distribution of CPI in the default server configuration and in the server provisioned based on the output of iBench, while Fig. 7c compares the CPU utilization in the two systems. In both cases accounting for contention when provisioning the system improves application performance (the CPI curve is shifted to the left in the new system) and reduces CPU throttling due to memory stalls. Similarly, we can use the information on resource contention to guide the microarchitecture design (cache hierarchy, pipeline organization, etc.) of hardware aimed to service a particular application.

C. Application Development/Testing

An important reason behind resource inefficiency is poor application design. Workloads are often written without sufficient considerations of sensible resource usage, resulting in unnecessarily bloated code, huge memory footprints, and high CPU utilization. This problem is even more prominent in DC workloads, which are often complex, multi-tier applications with several interdependent components. Despite the long testing periods devoted to these workloads, robustness and performance are typically the main optimization objectives, with resource-efficiency being less important. Here we show that using iBench to identify code regions that cause high contention not only improves efficiency by eliminating unnecessary resource consumption, but is also beneficial to performance by reducing resource contention.

For this purpose we start with an unoptimized data mining application that performs collaborative filtering on a large dataset of sparse data. The data are movie ratings from 180k users. Running the original version of the code, which relies on Singular Value Decomposition and PQ-reconstruction [34], [11] experiences very high contention in LLC capacity, bandwidth, L1 d-cache and L2 cache capacity and bandwidth, memory bandwidth and FP computation. Running the program to completion takes approximately 1.6h. The performance of the original code is shown in Fig. 8 (leftmost figure, first column). The second and third figures in the first column show the CPU and memory bandwidth utilization of the program (normalized to sustained memory bandwidth for the server). After detecting the points of contention using iBench, we optimize parts of the code to make better use of system resources. In the first code iteration we switch to SIMD operations using SSE4 [26]. As shown in the second column in Fig. 8 both performance and resource efficiency benefit. The boost in performance comes from leveraging spatial locality in matrix accesses, while the decrease in required resources comes from performing fewer operations on larger chunks of data and reducing the misses to the cache hierarchy. We now repeat the interference characterization for the new program. iBench again helps identify remaining inefficiencies in the code that induce resource contention. We progressively address these with optimizations such as reordering of operations to

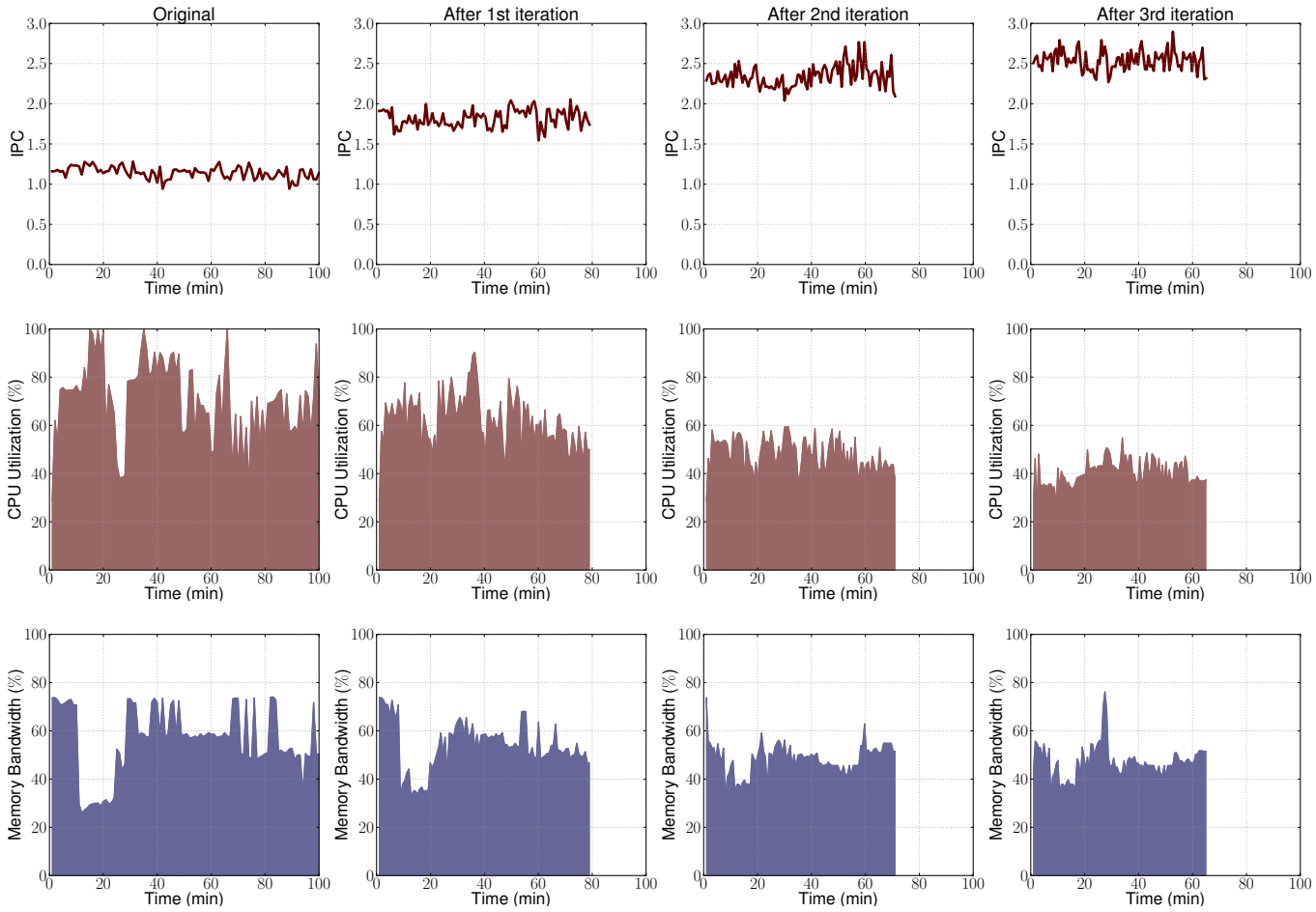

Fig. 8: Performance (IPC), CPU utilization and memory bandwidth utilization for the testing application across three optimization iterations using iBench. While performance for the original application is low, with the CPU being saturated, identifying contentious regions in the code progressively improves throughput and decreases resource utilization, improving resource efficiency.

the matrix elements or memoizing intermediate results. After each iteration we reevaluate the application's performance and resource utilization. As shown in the last column of Fig. 8 the final code runs in 35% less time than the original unoptimized version *while* requiring fewer system resources. While the code optimizations shown here are relatively straightforward, we believe that given the speed of obtaining the interference profile, using signals from iBench can significantly facilitate the development and testing process of large applications.

D. Scheduling in Heterogeneous CMPs

Finally, we show that iBench is applicable outside DC system studies. CMPs today consist of tens of - often - heterogeneous cores [12], [39], [44], [48]. Scheduling for these systems is challenging because in addition to the interference between applications that share resources, the scheduler should account for system heterogeneity. Similarly to the first use case, we design a simple scheduler that takes the interference profile obtained by iBench and identifies how each application from a multiprogrammed mix should be mapped to heterogeneous cores. When the mix is submitted to the system, each workload is briefly profiled against the iBench workloads to obtain its interference profile. Each SoI requires at most 10msec and runs can be done in parallel by replicating and sandboxing the application binary. Profiling can additionally leverage classification techniques to reduce the training overhead, by only profiling against a subset of SoIs and deriving the missing entries based on similarities with previous applications [9], [17]. We first create 40 4-SPECCPU2006 application mixes and schedule them on a simulated 4-core CMP [36] with 2 Xeon-like and 2 Atom-like cores from different generations each. The simulator captures contention in the cache and memory hierarchy, therefore the same process as before is used to quantify the impact of interference on application performance. The examined workloads do not exhibit storage or network activity hence we do not use the SoIs creating contention in those resources (SoI3-5). SPEC workloads are classified with regards to their cache demands as insensitive (n), friendly (f), fitting (t) and streaming (s), and mixes are created based on the methodology in [37]. Cores differ in their frequency, private cache hierarchy and microarchitectural details (e.g., pipeline, prefetchers, branch

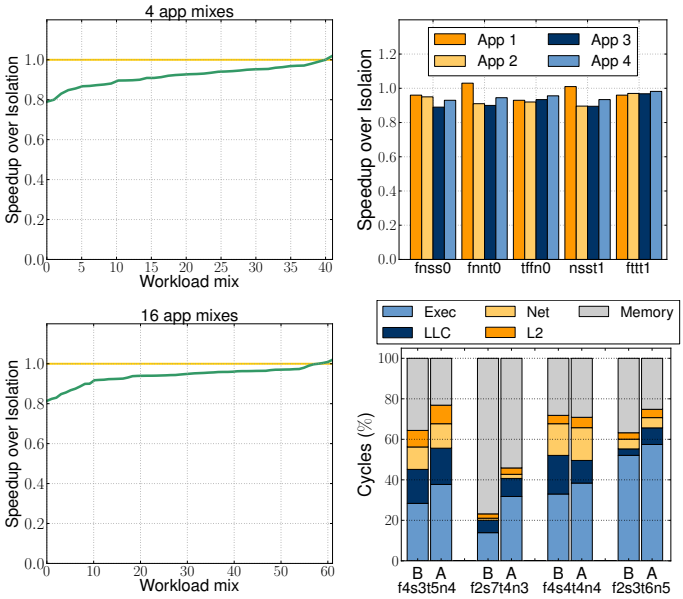

Fig. 9: Scheduling in heterogeneous CMPs. The upper figures show performance across the 4-application mixes and a per-application breakdown for selected mixes. The lower figures show the performance for the 16-application mixes and a breakdown of execution time to various subsystems for select mixes, before (B) and after (A) the use of iBench for scheduling.

predictors, issue width). All cores share an 8MB last level cache (LLC) and 16GB of memory. The scheduler uses iBench to identify the type of core and co-scheduled applications that constrain interference and selects the mapping that minimizes the average interference across workloads. Although this is not necessarily a global optimum it is good enough that performance does not degrade and utilization increases. The scheduler can also take advantage of workload signatures [12] to further refine the application-to-core mapping search space. We also create 60 16-application mixes and schedule them in a similar system with 16 cores. The variability in frequencies and cache hierarchies here is more widespread. Fig. 9 shows the performance obtained when using iBench to guide the scheduling decisions. The upper figures show the performance of the 4-app mixes ordered from worst to best-performing compared to isolated runs, and the breakdown to per-application performance for selected mixes. Performance degradations are marginal for most workloads. The lower figures show the performance across the 16-app mixes and the breakdown of clock cycles to the various subsystems for selected mixes. While without the use of iBench several mixes spend significant fractions waiting in memory instead of executing instructions, by minimizing interference larger fractions of time are devoted to useful execution. We plan to perform a more detailed study of scheduling tradeoffs in heterogeneous CMPs as part of future work.

VI. Conclusions

We presented iBench, a benchmark suite that measures the tolerated and caused interference of a workload in various shared resources. iBench is geared towards DC applications, but can also be applied to conventional workloads. It consists of 15 benchmarks (SoIs) that induce pressure over a wide range of shared resources that span the core, cache hierarchy, memory, storage and networking subsystems. iBench quantifies the type and degree of interference that an application generates in this set of shared resources. Similarly, it measures the type and intensity of interference an application can tolerate before violating its QoS across the same resources. We have validated the accuracy and consistency of iBench against a number of DC applications, ranging for conventional single-node applications, to distributed Hadoop workloads, and latency-critical online services. We have also evaluated a number of use cases for iBench. First, we use the interference information obtained with iBench to schedule workloads in an EC2-like environment in a way that minimizes interference between co-scheduled applications and improves system utilization. Second, we have shown how iBench can assist towards making informed decisions on the hardware specifications of a chip aimed for DC workloads, or on the provisioning of a DC server. Third, we have shown how iBench can be used by software developers to design more resource-efficient applications during testing. Finally, we have shown that iBench is applicable outside the context of DCs, and have used it for scheduling in large-scale heterogeneous CMPs. In all cases, using iBench significantly improves the decision quality and the performance, and resource efficiency of the system. We plan to release iBench under the GPL license, for researchers both in academia and industry to use.

Acknowledgements

We sincerely thank Daniel Sanchez and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. Christina Delimitrou was supported by a Stanford Graduate Fellowship.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/
[2] Apache Hadoop. http://hadoop.apache.org/
[3] Apache Lucene. http://lucene.apache.org/core/
[4] Apache Nutch. http://nutch.apache.org/
[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny. "Workload Analysis of a Large-Scale Key-Value Store". *In Proc. of SIGMETRICS, London, UK, 2012.*
[6] L. Barroso. "Warehouse-Scale Computing: Entering the Teenage Decade". *ISCA Keynote, SJ, June 2011.*
[7] L. A. Barroso, U. Holzle."The Datacenter as a Computer". *Synthesis Series on Computer Architecture, May 2009.*
[8] L. A. Barroso and U. Holzle."The Case for Energy- Proportional Computing". *Computer, 40(12):33–37, 2007.*
[9] R. M. Bell. Y. Koren, C. Volinsky."The BellKor 2008 Solution to the Netflix Prize". Technical report, AT&T Labs, Oct 2007.
[10] C. Bienia, et al. "The PARSEC benchmark suite: Characterization and architectural implications". *In Proc. of PACT, Toronto, CA, 2008.*
[11] L. Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". *In Proc. of COMPSTAT, 2010.*
[12] K. Craeynest, et al. "Scheduling Heterogeneous Multi-Cores through Performance Impact Estimation (PIE)". *In Proc. of ISCA, 2012.*

[13] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". *In Proc. of OSDI, SF, CA, 2004*.

[14] C. Delimitrou, C. Kozyrakis. "Cross-Examination of Datacenter Workload Modeling Techniques". *In Proc. of the First International Workshop on Datacenter Performance, DCPerf, Minnesota, MN*.

[15] C. Delimitrou, S. Sankar, K. Vaid, C. Kozyrakis. "Decoupling Datacenter Studies from Access to Large-Scale Applications: A Modeling Approach for Storage Workloads". *In Proc. of IISWC, Austin, TX, 2011*.

[16] C. Delimitrou, S. Sankar, A. Kansal, C. Kozyrakis. "ECHO: Recreating Network Traffic Maps for Datacenters of Tens of Thousands of Servers". *In Proc. of IISWC, San Diego, CA, 2012*.

[17] C. Delimitrou and C. Kozyrakis. "Paragon: QoS-Aware Scheduling in Heterogeneous Datacenters". *In Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, March 2013*.

[18] H. Esmaeilzadeh, E. Blem, et al. "Dark silicon and the end of multicore scaling". *In Proc. of ISCA, San Jose, CA, 2011*.

[19] Eucalyptus. http://www.eucalyptus.com/

[20] M. Ferdman, A. Adileh, et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". *In Proc. of ASPLOS, London, UK, 2012*.

[21] Google Compute Engine. https://cloud.google.com/products/compute-engine

[22] S. Govindan, J. Liu, A. Kansal, A. Sivasubramaniam. "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines". *In Proc. of SOCC, Cascais, Portugal, 2011*.

[23] J.R. Hamilton. "Cost of Power in Large-Scale Data Centers". http://perspectives.mvdirona.com

[24] A. Jaleel, M. Mattina, B. Jacob. "Last Level Cache (LLC) Performance of Data Mining Workloads On a CMP - A Case Study of Parallel Bioinformatics Workloads". *In Proc. of the 12^{th} HPCA, Austin, TX, 2006*.

[25] Y. Joo, V. Ribeiro et al. "TCP/IP traffic dynamics and network performance: A lesson in workload modeling, flow control, and trace-driven simulations". *In Proc. of SIGCOMM, 2001*.

[26] Intel Nehalem Architecture Optimization Reference Manual. April 2012.

[27] C. Kozyrakis, A. Kansal, S. Sankar, K. Vaid, "Server Engineering Insights for Large-Scale Online Services". In IEEE Micro, vol.30, no.4, July 2010.

[28] J. Mars, L. Tang, R. Hundt, K. Skadron, M. L. Soffa "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations". *In Proc. of MICRO, Brazil, 2011*.

[29] D. Meisner, C.M. Sadler, L. A. Barroso, W.-D. Weber and T. Wenisch. "Power management of online data-intensive services". *In Proc. of the 38th annual international symposium on Computer architecture, San Jose, CA, 2011*.

[30] Memcached. http://memcached.org/

[31] R. Narayanan, B. Ozisikyilmaz, et al. "MineBench: A Benchmark Suite for DataMining Workloads". *In Proc. of IISWC, San Jose, CA, 2006*.

[32] Open Compute Project. http://www.opencompute.org/

[33] Open Source Software for Building Private and Public Clouds. http://www.openstack.org/

[34] A. Rajaraman and J. Ullman. "Textbook on Mining of Massive Datasets", 2011.

[35] Roundcube. Open source webmail software. http://roundcube.net/

[36] D. Sanchez, C. Kozyrakis. "Fast and Scalable Microarchitectural Simulation of Thousand-Core Systems". *In Proc. of the International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, 2013*.

[37] D. Sanchez, C. Kozyrakis. "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning". *In Proc. of the International Symposium on Computer Architecture (ISCA), San Jose, CA, 2011*.

[38] S. Sengupta and R. Ganesan. "Workload Modeling for Web-based Systems". *In Proc. of CMG, 2003*.

[39] D. Shelepov, J. Saez, et al. "HASS: A Scheduler for Heterogeneous Multicore Systems". *In OSP, vol. 43, 2009*.

[40] L. Tang, J. Mars, M. L. Soffa. "Compiling For Niceness: Mitigating Contention for QoS in Warehouse Scale Computers ". *In Proc. of CGO, San Jose, CA, 2012*.

[41] VMWare Distributed Resources Scheduler. http://www.vmware.com/products/datacenter-virtualization/vsphere/drs-dpm.html

[42] VMWare vCloud. http://www.vmware.com/products/datacenter-virtualization/vcloud-suite/overview.html.

[43] Windows Azure. http://www.windowsazure.com/en-us/

[44] J. Winter, D. Albonesi. "Scheduling algorithms for unpredictably heterogeneous CMP architectures ". *In Proc. of DSN, 2008*.

[45] S. Woo, M. Ohara, et al. "The SPLASH-2 Programs: Characterization and Methodological Considerations". *In Proc. of ISCA, Santa Margherita, Italy, 1995*.

[46] XenServer. http://www.citrix.com/products/xenserver/overview.html

[47] X. Zhang, E. Tune, et al. "CPI2: CPU performance isolation for shared compute clusters". *In Proc. of EuroSys, Prague, Czech Republic, 2013*.

[48] T. Zidenberg, I. Keslassy, U. Weiser. "MultiAmdahl: How Should I Divide My Heterogenous Chip?" *In CAL, vol. 11, 2012*.