PABST: Proportionally Allocated Bandwidth at the Source and Target

Derek R. Hower
Qualcomm Technologies, Inc.
dhower@qti.qualcomm.com

Harold W. Cain   Carl A. Waldspurger
Qualcomm Datacenter Technologies, Inc.
{tcain, c_carlw}@qti.qualcomm.com

***Abstract*—Higher integration lowers total cost of ownership (TCO) in the data center by reducing equipment cost and lowering energy consumption. However, higher integration also makes it difficult to achieve guaranteed quality of service (QoS) for shared resources. Unlike many other resources, memory bandwidth cannot be finely controlled by software in existing systems. As a result, many systems running critical, bandwidth-sensitive applications remain underutilized to protect against bandwidth interference.**

In this paper, we propose a novel hardware architecture allowing practical, software-controlled partitioning of memory bandwidth. Proportionally Allocated Bandwidth at the Source and Target (PABST) precisely controls the bandwidth of applications by throttling request rates at the source and prioritizes requests at the target. We show that PABST is work conserving, such that excess bandwidth beyond the requested allocation will not go unused. For applications sensitive to memory latency, we pair PABST with a simple priority scheme at the memory controller. We show that when combined, the system is able to lower TCO by providing performance isolation across a wide range of workloads, even when co-located with memory-intensive background jobs.

I. Introduction

In the data center, workload consolidation (also called co-location) reduces the total cost of ownership (TCO) by performing the same amount of work with fewer machines [1], [2], [3], [4], [5], [6], [7]. However, shared resource contention on a consolidated machine can also reduce the performance of user-facing, latency-critical workloads [8], [6]. Applications may slow down enough to violate the service-level agreements (SLAs) that drive profit, leading operators to intentionally under-utilize machines (10-50% utilization [9]) despite the operational cost benefits [10], [2]. That intentional under-utilization will be increasingly difficult to justify going forward as core counts continue to rise (24-48 core single socket systems are now available or announced from Qualcomm Datacenter Technologies, Inc. [11], Intel [12], and Cavium [13]).

To achieve the cost benefits of consolidation without violating service agreements, providers must have the ability to provision shared resources to achieve performance isolation. Most of the resources on a multicore machine can already be provisioned to prevent workload interference using a combination of mature software technologies like virtual machines or container systems [14] and hardware technologies like virtual memory and cache partitioning [15], [16], [17]. Prior work has shown how careful management of these resources can increase consolidation without violating SLAs [10], [4], [18]. However, shared bandwidth, both on and off chip, remains a difficult resource to control, and the lack of hardware support "complicates and constrains the efficiency of any system that dynamically manages workload co-location" [10]. As a result, state-of-the-art data center management solutions choose either performance isolation or high utilization, but not both [10], [4], [19].

Many bandwidth provisioning schemes have been proposed in the literature. Generally, prior proposals fall into one of two categories: those that regulate bandwidth at the source (*i.e.*, CPU) by throttling request rates [20], [21], [22], [23], [24], and those that regulate bandwidth at the target (*i.e.*, the memory controller) by prioritizing queued requests [25], [26], [27], [28], [29]. We show in this paper that neither approach is sufficient by itself; **work-conserving bandwidth provisioning requires regulation at *both* the source and the target**.

Systems that regulate bandwidth at the target through priority arbitration are only effective if queues at the target can hold all outstanding requests. Otherwise, requests queue elsewhere in the system (*e.g.*, at the last level cache) where they are not subject to the arbitration. While the queuing requirement for target regulation may be feasible on small systems, a large data center system would need hundreds of queue slots at every memory controller in the system. Commodity memory controller request queues are an order of magnitude smaller.

Similarly, a work-conserving bandwidth allocation system that throttles requests at the source can only effectively provision bandwidth if workloads are insensitive to request latency. The amount of bandwidth a latency-sensitive workload can generate decreases as memory latency increases. Since a source-based system will not lower queuing delays at the target, it is ineffective at bandwidth regulation when latency-sensitive workloads are involved.

We illustrate the differences between source- and target-based regulation in Figure 1. The figure shows two different workloads. For the left two columns (a-b), we run two streaming applications and attempt to allocate bandwidth between the two streamers in a 3:1 ratio. For the right two columns (c-d), we run one pointer-chasing workload that is highly sensitive to memory latency with the same streaming application and same 3:1 ratio as before. We evaluate both workloads using a source- and target-based regulator, respectively, the details of which are found in Section IV.

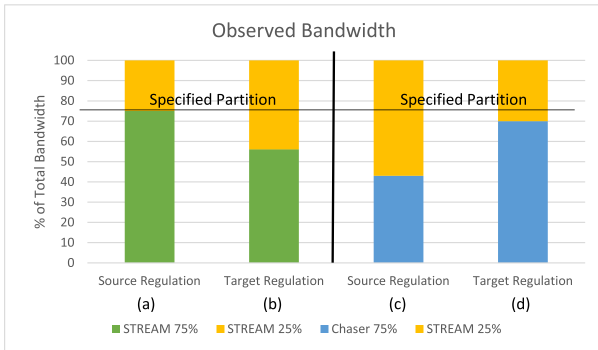

Fig. 1. Observed bandwidth consumption using either a source-only bandwidth regulator or a target-only bandwidth regulator. In all cases, one application is allocated 75% of the bandwidth and the other 25%. The best regulation approach depends on the workloads.

Figure 1 shows two clear results. First, target-based regulation is ineffective when running workloads that can generate enough concurrent requests to oversubscribe the target queues. In column (b), in which the two streaming workloads run together and flood the system with requests, the target-based regulator has a 76% allocation error. Second, source-based regulation is ineffective when the amount of bandwidth an application can generate depends on memory latency. In column (c), in which the pointer-chasing application is paired with a streaming application, the source-based regulator has a 128% allocation error because it is unable to lower the memory latency for the pointer-chasing workload without sacrificing utilization. In contrast, the target regulator, which lowers the latency for the chaser by reducing queuing delay, fares better in column (d). As we discuss in Section IV, the remaining 20% error from the target regulator can only be eliminated by sacrificing the efficiency of the memory controller.

Based on the prior observations, we propose PABST, a unified proportional bandwidth allocation mechanism that regulates bandwidth at both the source and target. PABST enforces a *proportional share* bandwidth allocation for each class of service. Proportional shares are a common means to specify differentiated QoS allocations in existing frameworks, including those widely used in the data center (*e.g.*, Linux cgroups [30], VMware [14]). By using proportional shares, PABST provides a hardware mechanism and leaves allocation policy up to software.

PABST uses two components that work together to provide robust bandwidth management. First, a bandwidth *governor* at the source proportionally throttles requests in response to a saturated memory controller. Second, a simple *arbiter* is added to the memory controller scheduler to prioritize the requests based the same proportional bandwidth share.

We show that PABST enforces bandwidth allocation with the following properties:

- **Proportional allocation:** When all QoS classes can generate at least their allotted share, the observed bandwidth will be in the same ratio as the allotted proportional shares.
- **Work conservation:** Excess bandwidth will not go unused if at least one QoS class can consume it.
- **Proportional distribution of excess bandwidth:** If any QoS class fails to consume its entire share, the excess bandwidth is distributed proportionally among the remaining classes.

When PABST is used as part of a comprehensive, multi-resource quality of service solution, users can simultaneously achieve a minimum level of service for critical workloads and maximize overall system utilization. Because PABST is work conserving, only those QoS classes that exceed their fair share of bandwidth are throttled. And, notably, that fair share is determined dynamically in response to system conditions to ensure the system runs near maximum utilization at all times. Even a class with the smallest proportional share is able to consume 100% of bandwidth when no other class competes for the resource.

We show that PABST is able to accurately allocate bandwidth across a wide range of workloads, ultimately eliminating the long tail latencies in service times, reducing bandwidth-induced slowdowns of high-priority workloads from 2.3–1.6× to 1.3–1.1×, and increasing the utilization of a consolidated host by up to 2*x*.

II. QoS Background

A bandwidth allocation mechanism like PABST is one piece of a robust Quality of Service (QoS) architecture. In this section, we first discuss two important use cases for a QoS architecture. Then we describe how software classifies software abstractions into service classes, and finally describe the proportional-share interface used by PABST.

A. QoS Use Cases

In this paper, we focus on two main use cases for a Quality of Service architecture.

Use Case 1: Performance isolation for high-priority services in consolidated systems. In a data center, a small number of services drive the majority of the profit [31]. These services, like search or ad auctions, often have strict performance requirements. However, despite their importance, these services rarely utilize available hardware resources completely, leaving slack that could be used by less important workloads to improve efficiency. Without the ability to guarantee performance isolation, data center operators will often conservatively schedule such high-priority workloads by themselves despite the fact that doing so often leads to lightly-utilized systems [2]. A QoS architecture could enable higher system utilization if it were able to control the impact low-priority jobs have on high-priority jobs.

To be effective for this use case, a bandwidth allocation mechanism needs to (1) be able to quickly provision bandwidth to meet the demand for a high priority service, and (2) reallocate any excess bandwidth to low priority services to increase utilization.

Use Case 2: Infrastructure as a Service (IaaS). Data center operators, such as Amazon Web Services, Google

Cloud, and Microsoft Azure sell their compute infrastructure as a service. Customers bundle their applications in a virtual machine (VM) and submit it to the cloud along with resource requirements like number of cores and cache capacity. As systems become increasingly bandwidth constrained, IaaS providers will likely add memory bandwidth as a resource, allowing customers to pay for the amount of bandwidth they need. In fact, Ben-Yehuda *et al.* argue that IaaS will evolve to become Resource as a Service where fine-grained resources are offered based on dynamic market pricing [32]. In either case, operators will need to provide at least the amount of bandwidth agreed upon. However, if excess bandwidth exists on the physical machine in the data center, it is in the operator's economic interest to distribute that bandwidth in order to increase the throughput of jobs and utilization of the data center as a whole; doing so allows them to sell more virtual machines with the same amount of hardware.

In the IaaS use case, a system may be running multiple VMs with similar priority and similar bandwidth allocations. To be effective, a bandwidth allocation mechanism should guarantee each VM gets at least its specified share. It should also redistribute excess bandwidth fairly among all VMs in order to increase utilization.

B. QoS Framework

PABST operates as piece of a larger QoS architecture. Such frameworks already exist in commercial products [17], and for PABST we only propose adding a new bandwidth allocation knob for each class of service.

A QoS architecture generally performs three functions: classification, monitoring, and allocation. For classification, a QoS identifier (QoSID) register is added to each CPU. Any two threads that have the same QoSID value belong to the same QoS class, and that QoS class serves as a container for resource monitoring and allocation. The QoSID is under the control of privileged software, which has the flexibility to create arbitrary groups to match various software abstractions. Common abstractions include virtual machines, containers, process groups, and applications.

QoS monitors allow users to measure the amount of each resource consumed by each class. For example, Intel's QoS architecture supports querying per-QoS-class last level cache occupancy and memory bandwidth. The monitoring features are useful when deciding how to best schedule workloads to minimize interference.

Existing QoS architectures provide separate allocation controls for each shared resource. In this paper, we assume that the baseline system already provides controls to partition shared caches among QoS classes, similar to Intel's Cache Allocation Technology [17]. We use those controls to isolate each class of service in the cache. We discuss the interplay between cache and bandwidth allocation in more detail in Section V.

For PABST, we add a single bandwidth *proportional share* allocation control for each class of service. The share parameter is set by the privileged software on behalf of administrators and users and is discussed in more detail next.

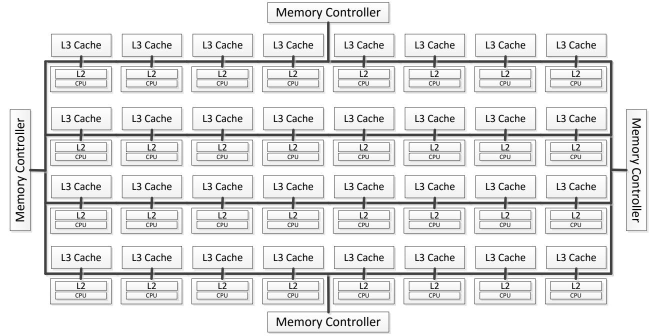

Fig. 2. Baseline: a 32-core, 4 memory controller mesh

C. Proportional Shares

In a proportional-share model, the fraction of resources allotted to a consumer i is expressed by assigning it a numerical $weight_i$ value. The system guarantees that within an interval of time, the consumer will have available no fewer than $weight_i/weight_{total}$ of the resource, where $weight_{total}$ is the sum of weights aggregated across all consumers.

$$ProportionalShare_i = \frac{weight_i}{\sum_{\forall j} weight_j} \quad (1)$$

While it is common for software and users to specify proportional shares in terms of weights, the PABST algorithm discussed later is simplified by using the inverse of weight, commonly referred to as a *stride* [33].

$$stride_i \propto \frac{1}{weight_i} \quad (2)$$

The stride represents the relative cost for a class to use a resource. Higher strides translate into a lower share of the resource. For example, a class given a stride of two will get half as much of the resource as a class given a stride of one.

Users request resource allocations based on the relative priority among the concurrently running applications. For example, a latency-sensitive application may be given a grossly disproportionate share to ensure that bandwidth consumption of low-priority background tasks do not negatively affect response time. In a virtualized system hosting virtual desktops, it may make sense to give each an equal share. Significant related work in this area exists, any of which could use PABST as the underlying bandwidth control mechanism [10], [34], [35], [36].

III. PABST

To keep the discussion concrete, we will describe PABST in the context of the baseline system shown in Figure 2. The system contains 32 CPUs arraigned in an 8x4 tiled SoC. All system components are connected via an on-chip high-bandwidth mesh interconnect. A tile contains a CPU, L1 instruction and data caches, a unified private L2 cache, and one slice of a shared L3 cache. The L3 cache supports way-based capacity partitioning that allows us to exclusively allocate a

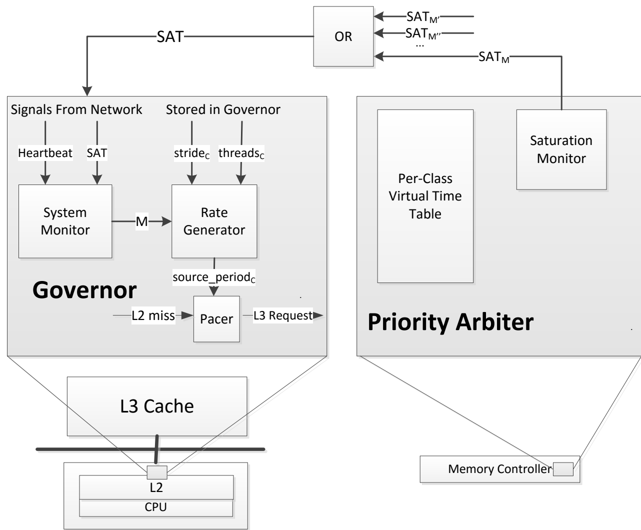

Fig. 3. PABST Components

TABLE I
PABST GOVERNOR STATE

M	A multiplier indicating the amount of throttling required to keep the memory controllers from overcommitting.
δM	The magnitude of the next change in M.
E	The number of consecutive epochs in which the direction (up/down) of the rate change has not switched
Phase	Encodes the current direction of the goal rate and δM, respectively.

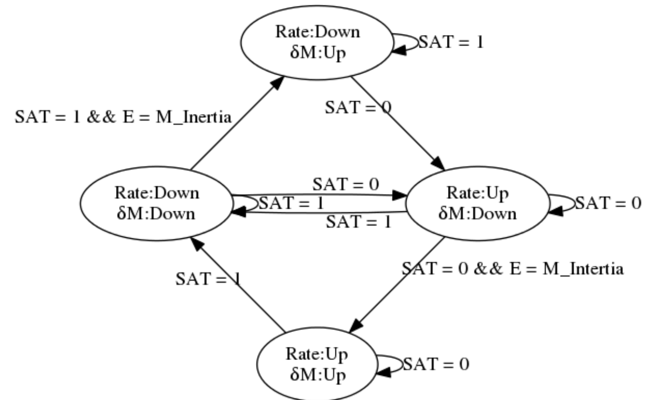

Fig. 4. State transition diagram for the algorithm calculating M. *M_Intertia* is a configurable parameter that trades off responsiveness and stability, and is set to 3 in the evaluation. A transition occurs at every epoch heartbeat and M is recalculated according the actions in Table II.

portion of the cache to a QoS class. Four memory controllers are located on the edges of the system.

A. Overview

PABST has two components, shown in Figure 3: a governor at each L2 cache (source) and a priority arbiter in the memory controllers (target). The governor throttles requests from the CPU in the same tile by pacing the rate that L2 cache misses can enter the SoC network. The rate is calculated using a feedback loop based on both the weighted share assigned to the CPU being throttled and the current bandwidth demand.

The priority arbiter is a simplified version of a fair earliest-deadline arbiter. The arbiter lowers the latency of DRAM accesses in proportion to the same per-class weighted share used by the governor. Together, the two components work in tandem to provide work-conserving bandwidth allocation.

B. Governor

The PABST governors all work in tandem to (1) allow enough traffic in the system so that the memory controllers are fully utilized but not oversubscribed and (2) ensure that the relative request rates at each CPU are in the same proportion as the assigned proportional weights. The governors work in lockstep to achieve the coordination but *do not communicate with one another*. Rather, the distributed algorithm that the governors run is designed to produce goal request rates in the correct proportions as long as all governors see the same two inputs: an epoch heartbeat and a binary saturation signal (SAT) indicating whether or not the memory controllers were oversubscribed during the previous epoch.

At each epoch boundary (*e.g.*, 10 μs), the governors decide whether to adjust the goal request rates higher or lower and by how much. In general, the governors raise the goal request rate when the saturation signal indicates that the memory controllers can handle more requests without being overcommitted and lowers the goal request rate otherwise. The magnitude of the adjustment depends on the recent saturation history; adjustments are larger when the saturation signal has been consistently high or low and smaller when the saturation signal fluctuates. This rate policy ensures that the governors are (1) quick to respond to changes in bandwidth demand and (2) insensitive to noise on the saturation signal during periods of stable bandwidth demand.

The governors consist of three subcomponents. The *system monitor* tracks how far the current bandwidth demand in the system is from the system's bandwidth capacity. The system monitor produces a multiplier, M, which is an indication of how fast the system needs to change in order to either relieve bandwidth pressure or increase demand to achieve higher utilization. Since the system monitor tracks global system behavior, all governors will independently produce the same M values. The *rate generator* translates the system-wide multiplier into a class-specific request rate. Those rates are always proportional to the user-specified class share, and the rates will sum across the system to so that the total bandwidth demand matches the target implied by M. The *pacer* control unit enforces the target rate by spacing requests coming from the source (L2 cache). The pacer operates over a sliding time window ($\ll$ the epoch length), which allows bursts of requests to proceed unthrottled as long as the class stays under its target rate within the window.

We discuss each subcomponent in detail below.

1) System Monitor: The state machine to calculate M is shown in Figure 4 and Tables I and II. M moves in the opposite

direction of the goal rate change (since it ultimately determines the period, which is inversely related to the rate), and always decreases following a low SAT signal and increases following a high SAT signal. This causes the system to drive more traffic when the memory controllers are under-committed and drive less traffic when the memory controllers are overcommitted.

The magnitude of the change in M, denoted by δM, is small when the SAT signal is noisy (implying the system is running near ideal bandwidth usage) and increases rapidly when SAT is steady so that PABST is responsive to sudden changes in bandwidth demand. To accomplish this, δM always decreases following a high SAT signal and only starts to increase again once the SAT signal has stayed low for a configurable number of epochs, which we call the *inertia* of δM (*e.g.*, 3). This hysteresis gives PABST stability without sacrificing too much responsiveness. Changes in δM are always exponential, and thus are implemented efficiently using a shift.

2) Rate Generator: The governor expresses the goal request rate as an average period, in cycles, between requests (*i.e., L2 misses*) that enter the SoC network. A pacer circuit, discussed in Section III-B3, uses the period to space out requests generated at the source. The request period is determined by scaling the multiplier M by the share of the class running behind the governor. Once a new value of M is generated, it is combined with the stride, $stride_c$, assigned to the QoS class c running on the CPU to produce the goal request period.

$$class_period_c = (M * stride_c)/F \tag{3}$$

The constant scale factor F, which effectively enables fractional rate changes, gives more precise control over bandwidth shares. If F is too high, then the algorithm is slow to converge. If F is too low, then the algorithm may be unstable since the period change between epochs may be too large. We find that a value of 16 (right shift by four) works well in practice for our baseline system.

Since strides are assigned to aggregate QoS classes rather than individual CPUs, the stride must be adjusted to account for the number of CPUs actively executing a class. Since we are dealing with periods, multiplying the stride by the number of active CPUs, denoted by $threads_c$, equally distributes the bandwidth allocation for a class among all active threads.

$$source_period_c = class_period_c * threads_c \tag{4}$$

We discuss in Section V how $threads_c$ can be tracked in the system.

Maintaining Proportional Shares Every rate change that a governor makes is proportional to the share of the QoS class it is trying to throttle. As a result, PABST maintains the invariant that the target rate is always in the same proportion as the share specified by $weight_c$:

$$\frac{Rate_c}{\sum_{\forall_j} Rate_j} = \frac{weight_c}{\sum_{\forall_j} weight_j} \tag{5}$$

Note that this invariant holds by design, and does not require any governor to know the target rate of any other governor in the system. Also note that the final target rate determined by the governor may be higher than the rate that the workload running on the CPU can actually generate. This happens, for example, when one CPU is able to saturate the memory controller bandwidth on its own while another CPU is in a phase that infrequently accesses the memory controller. This property makes PABST work conserving; a bandwidth hogging CPU is able to consume all the bandwidth while the other CPU doesn't need it.

3) Pacer: The request period calculated by the governor is enforced by a component called the pacer. The pacer enforces the target rate by calculating the next cycle a request can issue given the last request time and current period. The pacer builds credit during periods of idleness, allowing for request bursts to proceed unthrottled when the CPU has underutilized its bandwidth allotment in the recent past.

The pacer tracks two timestamps, C_{next} and C_{now}. C_{next} is the next cycle that the cache is allowed to issue a new request and C_{now} is the current time (cycle count). Requests are throttled whenever $C_{next} < C_{now}$.

When an L2 miss issues to the SoC network, the pacer updates C_{next} to $C_{next} + source_period_c$. The pacer builds credit as C_{now} and C_{next} get further apart. We do not want the credit to be unbounded, however, as that could allow a class to exceed its target rate within an epoch. Thus, we bound the credit by never letting C_{next} get more than N cycles behind C_{now}. In our evaluation, we select $N = stride_c * 16$, effectively allowing bursts of up to 16 requests to proceed before being throttled.

Accounting for Cache Filtering Looking at the system model in Figure 2, PABST throttles a private L2 cache, even though this component does not directly generate requests to DDR memory. Rather, the memory requests are actually generated from the shared L3 cache. To ensure that the target rates determined by the governor translate into the same bandwidth share at the memory controller, PABST must adjust the target rates to account for accesses that hit in the L3 cache.

Conceptually, we would like to scale the target rate by the ratio of local cache misses to shared cache misses. Calculating this ratio on the fly would be difficult, and so we instead use an approximation that is equivalent when viewed over a modest window of time.

When a request leaves the local cache, we adjust C_{next} assuming that the request will find its way to memory. If the request is instead serviced by the shared cache, we undo the C_{next} update by subtracting, rather than adding, the current stride.

The shared cache may also generate a writeback if the incoming demand request causes a replacement. That writeback causes extra memory bandwidth, which we ultimately account to the same class as the demand request[1]. To do so, the shared cache notifies the governor, via a flag on the response message, when a writeback occurs. On receiving the writeback

[1]See Section V for a discussion of other writeback accounting methods.

TABLE II
PABST GOVERNOR STATE MACHINE TRANSITION ACTIONS.

Phase	Rate:Up; δM:Up	Rate:Up; δM:Down	Rate:Down; δM:Down	Rate:Down; δM:Up
Rate:Up **δM:Up**	$M = M - \delta M$ $\delta M = \delta M * 2$ $E = E + 1$	-	$M = M + \delta M/2$ $\delta M = \delta M/4$ $E = 0$	-
Rate:Up **δM:Down**	$M = M - \delta M$ $\delta M = \delta M * 2$ $E = E + 1$	$M = M - \delta M$ $\delta M = \delta M/2$	$M = M + \delta M/2$ $\delta M = \delta M/4$ $E = 0$	-
Rate:Down **δM:Up**	-	$M = M + \delta M/2$ $\delta M = \delta M/4$ $E = 0$	-	$M = M - \delta M$ $\delta M = \delta M * 2$ $E = E + 1$
Rate:Down **δM:Down**	-	$M = M + \delta M/2$ $\delta M = \delta M/4$ $E = 0$	$M = M + \delta M$ $\delta M = \delta M/2$ $E = E + 1$	$M = M + \delta M$ $\delta M = \delta M * 2$

notification, the pacer updates C_{next} as if that writeback were a demand request (*i.e.*, by adding the stride).

C. Memory Controller

The baseline memory controller is split into two logical partitions. A front-end accepts new requests from the SoC network and places them in either a read or write request queue. The back-end schedules accesses to the DRAM banks. Requests flow from the front-end to the back-end greedily, with preference given to reads over writes.

We add two functions to the baseline memory controller: a saturation monitor and a priority arbiter.

1) Saturation Monitor: The saturation monitor calculates the average occupancy of the front-end memory controller read queue over the prior epoch. When the average occupancy is greater than half of the queue capacity, the SAT signal for the memory controller is raised. The SAT signals from all memory controllers are logically ORed together and then are forwarded to the governors at epoch boundaries. If traffic is unevenly distributed to the memory controllers, the combined SAT signal may lead to one or more underutilized memory channels. This potential problem can be mitigated at least two ways: with a uniform address hash that evenly distributes requests to the memory controllers (our assumption in the evaluation) or by throttling requests to each memory controller independently. The latter solution would require a SAT signal per memory controller and a governor per memory controller.

2) Priority Arbiter: The priority arbiter tracks the bandwidth usage of each class and prioritizes requests from classes that are furthest behind their target consumption. By prioritizing requests, the arbiter effectively lowers the latency of requests from high-priority classes. In general, the priority arbiter only affects the selection of read requests; since write requests are not on the critical path, their selection order does not affect performance. We leave the baseline channel read/write switching policy unmodified.

The arbiter maintains a virtual clock per QoS class. The virtual clock is incremented by $stride_c$ whenever a new request is accepted. Upon entering the memory controller queues, a request is assigned a virtual deadline equal to the current virtual time of the class. To avoid an unbounded amount of virtual credit from building during periods of idleness, the virtual deadline is capped be at most *slack* (*e.g.*, 128) virtual ticks behind the last virtual deadline picked by the arbiter. If a virtual deadline is limited by *slack*, then the updated value is also written into the virtual clock for the class.

When selecting a read, the front-end arbiter selects a ready request that has the earliest virtual deadline. Write requests are not prioritized since they are not on the critical path. The back-end arbiter prioritizes row hits over other ready requests, which are serviced in virtual deadline order. This is a fair variant of First-Ready, First-Come-First-Serve scheduling [26].

The arbiter is similar to the Fair Queuing Memory (FQM) System proposed by Nesbit *et al.* [26], with several changes:

- The PABST arbiter tracks true virtual time by incrementing a per-class virtual clock by $stride_c$ for each request. FQM approximates virtual time by scaling an expected access time, in real clock cycles, by a per-class weight.
- While FQM charges more for requests that take longer (*e.g.*, bank conflicts), the PABST arbiter charges the same amount of virtual time per access. Our modeling showed the single charge method to be equally effective, perhaps because we use a closed-page policy.
- We apply earliest-deadline-first priority in two places: a memory controller front-end queue and the memory controller back-end (bank) queues.
- We do not track virtual deadlines for banks and channels separately. Instead, the controller only schedules and prioritizes banks when the channel has free cycles.

D. Implementation Considerations

We assume that a centralized synchronizer sends out the epoch heartbeat at regular intervals on a dedicated wires and that the SAT signal is implemented as a global wired-OR line. Alternatively, dedicated wires could be avoided by negotiating epoch boundaries via sending special network packets on the interconnect, similar to coherence messages.

Note that starting epochs at the *exact* same point in real time for all governors is not critical, as long as the maximum lag between the start times is much smaller than the epoch size (the short period with the "incorrect" target rate will be in the noise when averaged over the length of the epoch). In other words, "lockstep" doesn't imply perfectly synchronous at the timescale of a single cycle, but rather at a timescale that

TABLE III
BASELINE SYSTEM PARAMETERS

CPU	2.2 GHZ, 32-entry LSQ
L1 I-cache	32KB, 64B lines, 8-way
L1 D-cache	32KB, 64B lines, 8-way
L2 cache	unified, 256KB, 8-way, 128B lines, private
L3 cache	40 MB (1.25MB / tile), 20-way, 128B lines, shared
Memory controllers	DDR4-2400 timing, closed adaptive page policy [39]

is a small fraction of an epoch. A similar approach was used in Martin *et al.*, to track distributed logical time [37].

The governor algorithm can be implemented using only additions and shifts and a multiply on small (*e.g.*, 12-bit) values, thus keeping the logic small and efficient. Also, the governor algorithm runs infrequently and is not on a critical path; in fact, the algorithm could even run on a slower clock than the cache.

The pacer, on the other hand, must run at the frequency of the cache since it must be able to match the maximum request rate. If the pacer circuit cannot complete in a single cycle, it could be pipelined. If pipelined, requests may take an extra cycle to be NACK-ed by the governor.

IV. EVALUATION

A. Methodology

To test PABST, we use an in-house, cycle-approximate simulator that models a data center-class server. We simulate the full system by using QEMU [38] as a functional front-end that drives the cycle-approximate model back-end. The guest image inside QEMU runs an unmodified Linux 3.15 kernel.

We model an out-of-order but non-speculative CPU that uses perfect branch prediction and perfect memory disambiguation. However, we enforce all register and address dependencies and model structural hazards, such as the Re-Order Buffer (ROB) size and Load/Store Queue (LSQ), that limit the instruction window. We combine the CPU model with detailed models of cache and DRAM, resulting in high fidelity on workloads bottlenecked by the memory system. We have validated that the performance reported by our simulator is within 10% of data center hardware when running memory-bound workloads (such as those evaluated in this paper).

The system under test is the 32-core SoC described in Section III with details outlined in Table III. We accurately model the bandwidth of all DDR DRAM and caches. Our DDR model is based on one from gem5 [40], [39], but is modified to support separate front-end and back-end queues. The cache models were developed in-house. We do not model internal SoC interconnect bandwidth, under the assumption that it is appropriately provisioned to handle traffic when running the memory controllers at peak throughput.

We investigate the properties of PABST using a mix of microbenchmarks, SPEC CPU 2006 applications, and a memcached server. We focus on two microbenchmarks: one that is limited by latency and one that is limited by bandwidth. The latency-sensitive microbenchmark (`chaser`) performs four independent random pointer chases on each CPU. When run on an out of order machine, `chaser` is able to sustain four concurrent memory requests, making its performance sensitive to both memory latency and bandwidth when multiple threads of `chaser` run together. We run multiple concurrent threads of `chaser` so that the benchmark can generate enough bandwidth to saturate the system when run in isolation. The bandwidth-sensitive microbenchmark (`stream`) is a hand-optimized program that streams through an array at a 128-byte stride. All the loads and/stores in `stream` are independent, meaning that its performance is limited only by the available bandwidth.

We use SPEC CPU2006 workloads as a proxy for data center applications, since prior work has shown that they degrade similarly to Google data center workloads in response to resource contention [2], [7], [6]. We only show the subset of the SPEC workloads that generate enough memory traffic to saturate memory bandwidth in the system when running on all CPUs, namely `GemsFDTD`, `lbm`, `libquantum`, `mcf`, `milc`, `omnetpp`, `soplex`, and `sphinx3`. We profiled each SPEC workload to collect simpoints [41], and run the highest weighted simpoint from each of the aforementioned workloads for 100 million instructions. When running N copies of the same SPEC workload together, we report the weighted slowdown (inverse of weighted speedup [42]), defined as

$$WeightedSlowdown = \frac{IPC^{SP}}{\sum_{i=0}^{N} IPC_i^{MP}} \qquad (6)$$

Due to infrastructure limitations, we run `memcached` on an 8-core system where all components are scaled down 4× compared to the 32-core system. The client thread runs in the same guest QEMU image as the server, but we drop all instructions coming from the client in the model so that it does not affect the reported performance. In all reported results, we run a single server thread for 1K transactions after warming up for 10K transactions.

Unless otherwise stated, classes sharing the system are given exclusive partitions of the last-level cache in order to isolate the effects of bandwidth sharing. When comparing to a baseline, we ensure that workloads in the both the baseline and experimental runs are given the same cache allocation. We use an epoch size of 10μs in all results.

B. Confirm Principles

First, we seek to confirm the principles laid out in Section I.

Principle 1: Proportional Allocation

In Figure 5, we show an execution with two QoS classes. Both classes are running the read `stream` microbenchmark on 16 cores. We allocate strides in a 7:3 ratio, such that one class should receive 70% of the memory bandwidth and the other 30% of the memory bandwidth. As shown in Figure 5, PABST is able to quickly find the target rates that result in the desired bandwidth allotment. Once at those rates, the bandwidth remains steady, with only small perturbations caused by operating system activity.

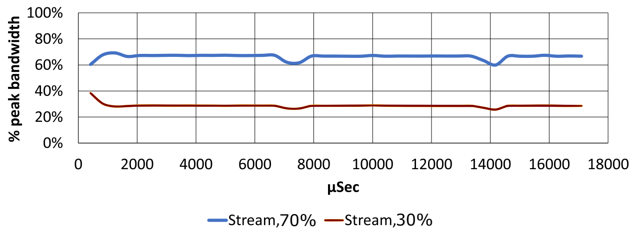

Fig. 5. Bandwidth consumed by two streaming microbenchmarks

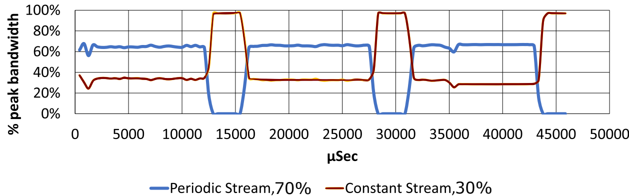

Fig. 6. Bandwidth consumed by a constant streamer and periodic streamer

Principle 2: Work Conservation / Do No Harm

To test that PABST is work conserving, we pair the same constant read `stream` used in Figure 5 with another streamer that cycles through periods of memory-resident and cache-resident streams. If PABST is work conserving, we would expect the constant streamer to consume extra bandwidth when the periodic streamer stops accessing memory. In our experiment, we allocate 70% of bandwidth to the periodic streamer and 30% of bandwidth to the constant streamer.

Figure 6 shows that PABST is able to quickly adapt the target rates in response to a bandwidth change. When the periodic streamer stops accessing memory, the constant streamer consumes nearly 100% of the system bandwidth. When the periodic streamer resumes accessing memory, the constant streamer is quickly throttled back to its 30% target allocation.

Principle 3: Proportionally Distribute Excess Bandwidth

We show how PABST distributes excess bandwidth in Figure 8. This experiment consists of one `stream` workload (L3 Stream) that fits in the L3 cache and two `stream` workloads (DDR Stream) that do not. The L3 streamer is allocated 25% of the memory bandwidth even though it does not significantly contribute after warming up. We should see that the bandwidth allocated to the L3 streaming workload is split in proportion to the allocation of the other workloads. In this particular case, we expect to see the high priority DDR stream, which is given a 50% allocation, get twice as much of the excess as the low priority DDR stream, which is given a 25% allocation. This is indeed the case: the high priority stream receives about 66% of the bandwidth, or 16% over its fair share, and low priority stream gets 33% of the bandwidth, or 8% over its fair share.

C. Source and Target Regulation

In this experiment, we seek to confirm that PABST achieves the benefits of both source and target regulation. We repeat the experiment shown in Figure 1, this time adding PABST

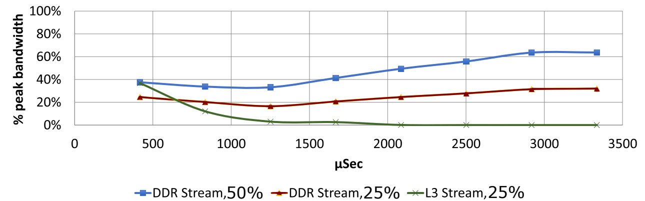

Fig. 7. Excess bandwidth is proportionally distributed

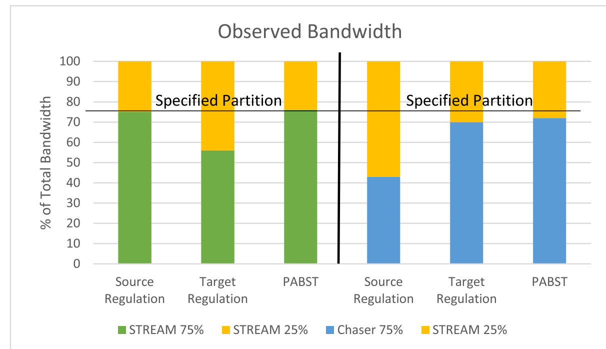

Fig. 8. PABST acheives the benefits of both source- and target-based regulation.

as a bandwidth allocation mechanism. The three left bars show the actual bandwidth consumption of two write `stream` classes with a 3:1 ratio allocation ratio being managed with either source-only, target-only, or both (PABST) regulators. The three right bars show the same experiment except the streamer with a higher allocation is replaced with the latency-sensitive `chaser` microbenchmark.

PABST tracks the regulation method that results in bandwidth consumptions closest to the specified ratio of 3:1. When the `chaser` microbenchmark is used, we see that PABST still has a small error in the bandwidth actual bandwidth consumption. This is because the PABST target priority arbiter has not lowered the request latency for `chaser` enough for it to generate 75% worth of bandwidth. This highlights a fundamental tradeoff in the priority arbiter – to lower the request latencies further, we would have to sacrifice some efficiency in the memory controller by using a request schedule that would lead to more unused cycles on the data bus.

D. Performance Isolation

The primary goal of PABST is to enable performance isolation in a consolidated environment. Towards that end, we investigate the impact of PABST on both transaction-oriented and batch-oriented jobs.

In Figure 9, we show how PABST can improve the service times of memcached [43] transactions when co-located with the `stream` microbenchmark. In this case, we consider memcached to be high priority and therefore give it a large proportional share relative to the streamer (20:1). The results show that PABST nearly eliminates both the average service time degradation and long tail. In our results, some ($< 5\%$)

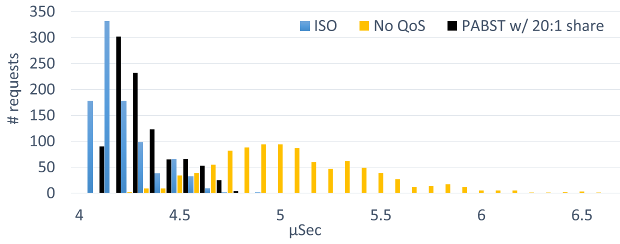

Fig. 9. Histogram of request service times for a memcached server running in isolation, with a background `stream` job with and without PABST.

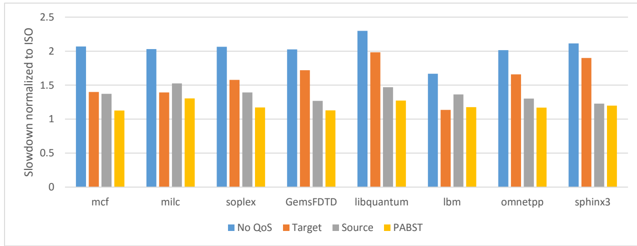

Fig. 10. Performance isolation with PABST. We show the slowdown of a high priority application (the SPEC workload) when paired with a streaming aggressor.

transactions in both the baseline and experimental runs that take orders of magnitude longer to complete than what is shown in Figure 9. The extreme scale of these service times point to system-level causes such as a page fault or disk access. We assume these types of outliers would be removed by highly-optimized software [8].

In Figure 10, a multiprogrammed workload from SPEC CPU running on 16 cores is paired with the read streamer benchmark running on the remaining 16 cores. The baseline is an execution with 16 SPEC applications in isolation (but with the same limited cache allocation). We allocate bandwidth in a 32:1 ratio to reflect disproportionate priority.

Overall, PABST reduces the slowdown induced by a bandwidth aggressor from an average of 2.0× to just 1.2×. PABST performs equally well on workloads that are mainly bandwidth limited (*e.g.*, `libquantum`) and workloads that are mainly latency limited (*e.g.*, `sphinx3`). We also show the performance of PABST when just the source governor or just the target arbiter are enabled. The two components complement each other, and the combination is always best.

E. Work-Conserving Fairness

PABST can also be used to provision a minimum portion of the memory system in a setting where classes have more comparable priority, as one might find in a consolidated IaaS scenario. While always providing a minimum memory service level corresponding to either a fair equal allocation or differentiated charge-back, PABST will also re-allocate the memory bandwidth during periods of underutilization.

We test the efficacy of PABST by simulating a consolidated IaaS environment. To model a machine with four virtual clients, we run four classes of service on the same machine.

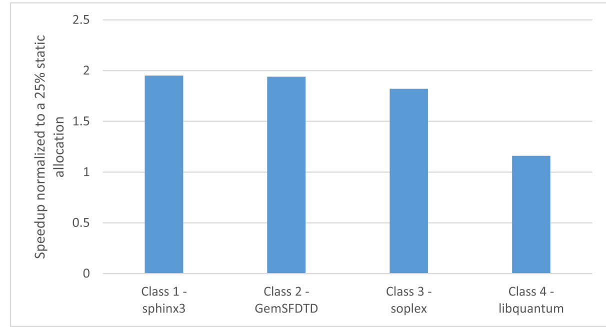

Fig. 11. Speedup of each QoS class running with a 25% allocation managed by PABST normalized to the performance of the class running with a static 25% bandwidth provision. The performance of each class improves because PABST is able to redistribute excess bandwidth.

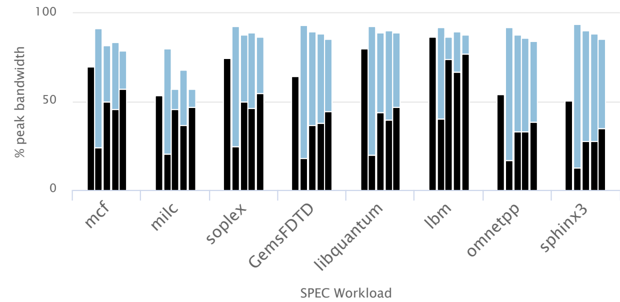

Fig. 12. Bandwidth consumed by SPEC workloads and a background streamer, respectively. SPEC bandwidth is in black (bottom) and STREAM bandwidth is in blue (top). Within each group we show, from left to right, SPEC in isolation, no bandwidth regulation, target-only regulation, source-only regulation, and finally PABST.

Each class uses 8 cpus, and within a class we run the same SPEC workload on all CPUs. We give each class a 25% allocation to represent an equally distributed resource.

We compare the performance of each class to an 8 CPU isolated run of the same workload but with DDR frequency scaled down 4×. This approximates the performance of the workload if it were to receive a static 25% allocation of bandwidth. In Figure 11, we see that the SPEC workloads achieve a 15-90% performance improvement compared to a static bandwidth allocation. This improvement arises because PABST is work conserving, and can redistribute excess bandwidth when needed.

F. Memory Efficiency

PABST reduces the overall memory efficiency of the system for two reasons. First, by prioritizing requests in the memory controller based on proportional shares, we limit the controller's ability to select the most efficient schedule. Second, the source governor algorithm transitions through several epochs where traffic is intentionally driven below the ideal saturation rate in order to discover what that saturation rate is for the current conditions.

Figure 12 quantifies the loss of efficiency when running SPEC workloads with a background streaming aggressor in

a 32:1 weight ratio (same workload mix as in Figure 10). To separate the two sources of efficiency loss, we show configurations where we have enabled only the governor, enabled only the memory controller priority, or enabled both.

First, we note that without any QoS support, the memory efficiency is typically high, largely due to the near-perfect memory-level parallelism of the stream microbenchmark. A notable exception is `milc`, which generates a request stream that is difficult to schedule efficiently in the memory controller. With bandwidth QoS enabled, the efficiency drops. The magnitude of the efficiency drop is correlated with the latency sensitivity of the SPEC workload. Workloads that get most of the performance isolation from the target regulator drive efficiency down the most. When a workload is sensitive to latency, it goes through periods of low memory-level parallelism. During these periods, there are few requests from that workload to choose from in the memory controller. When the memory controller is forced to pick among those few requests because the workload is high priority, we increase the chance that it will select a highly inefficient request.

Figure 12 also shows that for complex workloads, the proportional share of class may not translate into a percentage of the total bandwidth, at least not as one might naively expect. For example, `mcf` in isolation can drive about 70% of the peak memory bandwidth. While one may expect that `mcf` can drive that same amount of bandwidth when given any proportional share greater than 70%, we see that even when given a share of 32:1 (97%), the actual bandwidth usage drops. That drop is an indication that `mcf`'s ability to drive bandwidth has decreased because the average latency of memory accesses has increased due to the streamer. Though we have added the QoS priority scheduling at the memory controller to alleviate the problem, in general it is not possible to get back to the original isolation latency without giving up on work conservation.

V. Discussion

A. Setting Stride Values

The absolute magnitude of stride values impacts the performance of the PABST governor. If stride values are too large, the overall rate change between epochs may be too large to precisely hone in a target rate that maximizes system throughput. We mitigate this problem somewhat by using fixed-point arithmetic in the rate calculation (Equation 4), but extremely large strides will still cause large rate swings. Large stride values can also result in noticeable oscillations between epochs, giving the appearance of instability.

B. Tracking Active CPUs Per Class

Because PABST proportional shares are set per QoS class but the source pacers throttle individual CPUs, the governors scale the stride parameter by the total number of active CPUs executing the class (Equation 4). That means that either the hardware or system software must track the active CPU counts. In our evaluation, we have assumed that hardware tracks the active CPU counts for each class by updating a memory-mapped system register whenever the QoSID register changes. Updates are broadcast so that all CPUs in the class are notified of the new count. This broadcast mechanism is similar to the broadcast that must occur on all TLB invalidate instructions in the ARM architecture [44].

There may often be scenarios where some threads in a QoS class generate significantly more bandwidth that others. The current thread scaling mechanism, however, evenly distributes the bandwidth allocation among all threads in a class. In the future, we may consider adding feedback to facilitate heterogeneous allocations within a class.

C. L3 Writebacks

In Section IV, all of our experiments use L3 cache partitioning to isolate QoS classes from one another in the shared cache. If two or more QoS classes are able to share cache lines in the L3, a bandwidth algorithm like PABST must determine which class is responsible for evictions that cause memory write bandwidth. The answer is not obvious, and will often vary based on the situation.

Consider two conceptual applications: one, called `L3Res`, that fits entirely in the L3 and one, called `ReadStream`, that streams through DDR but performs no writes. When either of these runs in isolation, the L3 cache does not generate any writebacks, and therefore the memory controller sees no writes. When run together, `ReadStream` will cause evictions of `L3Res`'s dirty data, causing writebacks to the memory controller. `L3Res` will also cause writebacks of its own data to the memory controller, since it is no longer L3 resident when competing with `ReadStream`.

In this scenario, who is responsible for these writebacks? The system could charge a pre-determined class (*e.g.*, always `L3Res` or `ReadStream`) based on a notion of priority. It could also charge the class that allocated the line being evicted, or charge the class whose incoming request caused the eviction. These later two choices, while more dynamic, are also inherently unpredictable, and it is such unpredictability that leads to SLO violations and ultimately underutilization in consolidated data center nodes. For that reason, we believe that bandwidth allocation for QoS will always need to be paired with corresponding cache capacity QoS in order to properly account for such writeback traffic.

VI. Related Work

The ability to allocate memory system bandwidth to software tasks is critical for managing quality of service in modern processors. Unfortunately, hardware bandwidth controls are not yet provided by any commercially-available systems. As a result, state-of-the-art software resource managers like Heracles [10] must resort to coarse-grained alternatives, such as limiting the number of tasks allowed to run concurrently in a consolidated system, or isolating workloads on separate NUMA nodes in multi-socket systems.

Several prior hardware proposals for controlling memory system bandwidth rely on target-based quality-of-service mechanisms. Nesbit *et al.* proposed a fair queuing memory

system based on concepts from networking scheduling algorithms [26], and many follow-ons built on their work [27], [28], [45]. A key limitation is that allocations are enforced only among requests queued at the memory-controller target; research has demonstrated that this can lead to incorrect priority when the system is saturated with more requests than the memory controller can hold [21], [20]. Our own evaluation in Section IV confirms this result.

METE [45] uses a target-based mechanism for allocating memory bandwidth as one part of a comprehensive, multi-resource approach to system-wide quality of service. Their approach is complementary to PABST, which could be used in place of a target-only scheme for bandwidth allocation.

There are also many prior proposals in the literature for throttling memory bandwidth at the source. However, to the best of our knowledge, none of them set throttling rates based on a proportional shares interface that gives users fine-grained control over bandwidth as a resource.

Jahre and Natviq [21] and Ebrahimi *et al.* [20] both throttle bandwidth at the source in order to achieve fair slowdown in a consolidated system. Jahre and Natviq throttle bandwidth by limiting the number of Miss Status Holding Registers (MSHR) a class can occupy. Ebrahimi *et al.* pair an MSHR limit with a static rate limiter. Both proposals use a feedback loop driven by interference monitors that track how classes are contending with one another, and attempt to throttle bandwidth and cache capacity so that classes of equal weight slow down equal amounts. For many workload mixes, a fair slowdown utility function may lead to undesirable behavior; for example, a cache allocation that leads to thrashing may slow down workloads equally but also reduces system throughput. In contrast, PABST gives users the ability to treat memory bandwidth as a partitionable resource and leaves it to software to determine the appropriate allocation policies, which is itself a well-studied area [30], [14], [46]. PABST does not rely on large interference tracking tables.

MITTS takes the interesting approach of throttling bandwidth for a class by fitting traffic to a specific request distribution [24]. In doing so, they are able to achieve the same average bandwidth as a more static throttling mechanism but at improved performance since they allow bursty traffic to proceed without waiting. PABST does not shape traffic to a specific distribution but the pacer does allow bursts to proceed unthrottled by using stored credit. PABST unlike MITTS, is also work conserving, leading to better system utilization.

Herdrich *et al.* throttle bandwidth at the source through clock modulation and dynamic voltage and frequency scaling [23]. Their solution is implementable with today's hardware. However, it is not work conserving and can only be set at coarse intervals. Several software managers for consolidated environments have used Herdirch's approach, but all have lamented the lack of more fine-grained control [10], [4].

Muralidhara *et al.* [47] partitioned bandwidth by restricting classes to a subset of memory channels. In contrast to PABST, partitioning based on channels only works with a small number of classes and coarse allocations.

An alternative source-throttling approach uses feedback based on measured latency to manage access to congested resources. Delay-based network congestion control, pioneered by TCP Vegas [48], was extended in FAST TCP [49] to provide proportional-share control. PARDA [46] employs a similar technique for enforcing proportional-share fairness among distributed hosts accessing a storage array, adjusting queue lengths based on the ratio of observed latencies to a specified threshold. Instead of relying on fixed latency thresholds, TIMELY [50] utilizes latency gradients to adjust transmission rates in datacenter networks, in order to reduce both congestion and tail latency.

VII. Conclusions

We have shown that memory bandwidth contention leads to significant performance loss in co-located workloads. For that reason, systems often go intentionally underutilized in order to guarantee service requirements are met. To enable higher utilization without sacrificing performance, we have proposed the PABST system as a way to proportionally allocate bandwidth among classes of service. PABST uses both a source-based governor to throttle traffic in the system in conjunction with a simple endpoint-based priority scheme to lower the response latency of high-priority requests.

We have shown that together, the source-based governor and endpoint priority are able to isolate the performance of both latency- and throughput-oriented workloads. Performance isolation enables further workload consolidation in the data center and an overall reduction in total cost of ownership.

References

[1] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 127–144.

[2] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, pp. 248–259.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, 2011, pp. 295–308.

[4] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy proportionality and workload consolidation for latency-critical applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15, 2015, pp. 342–355.

[5] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 351–364.

[6] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011, pp. 283–294.

[7] L. Tang, J. Mars, and M. L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, pp. 1–12.

[8] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[9] L. A. Barroso, J. Clidaras, and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Morgan and Claypool Publishers, 2013.

[10] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Intl. Symp. on Computer Architecture*, 2015, pp. 450–462.

[11] Qualcomm, "Meet Qualcomm Centriq 2400, the worlds first 10-nanometer server processor," Press release, December 2016.

[12] Intel, "Intel Xeon processor E7-8890 v4 data sheet," June 2016.

[13] Cavium, *ThunderX Family of Workload Optimized Processors*, 2015.

[14] VMware, Inc., "vSphere Resource Management," https://pubs.vmware.com/vsphere-60/topic/com.vmware.ICbase/PDF/vsphere-esxi-vcenter-server-60-resource-management-guide.pdf, 2015.

[15] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. of the 40th Intl. Symposium on Computer Architecture*. ACM, 2013, pp. 308–319.

[16] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor e5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, March 2016, pp. 657–668.

[17] Intel, *Intel 64 and IA-32 Architectures Developer's Manual Vol 3B, chapter 17.4*.

[18] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014, pp. 4:1–4:14.

[19] K. Sudan, S. Srinivasan, R. Balasubramonian, and R. Iyer, "Optimizing datacenter power with memory system levers for guaranteed quality-of-service," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12, 2012, pp. 117–126.

[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 *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010, pp. 335–346.

[21] M. Jahre and L. Natvig, "A light-weight fairness mechanism for chip multiprocessor memory systems," in *Proceedings of the 6th ACM Conference on Computing Frontiers*, ser. CF '09, 2009, pp. 1–10.

[22] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, and D. Newell, "Pirate: QoS and performance management in CMP architectures," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 4, pp. 3–10, Mar. 2010.

[23] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses, "Rate-based QoS techniques for cache/memory in CMP platforms," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09, 2009, pp. 479–488.

[24] Y. Zhou and D. Wentzlaff, "MITTS: Memory inter-arrival time traffic shaping," in *Proceedings of the 43rd International Symposium on Computer Architecture. ISCA*, vol. 16, 2016.

[25] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. IEEE, 2010, pp. 1–12.

[26] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *Proc. of the 39th Intl. Symp. on Microarchitecture*, 2006, pp. 208–222.

[27] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, December.

[28] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '07, 2007, pp. 245–258.

[29] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 39–50.

[30] P. Menage, *CGroups Users Guide*, available on www.kernel.org.

[31] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 158–169.

[32] M. Ben-Yehuda, O. Agmon Ben-Yehuda, and D. Tsafrir, "The nom profit-maximizing operating system," in *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '16, 2016, pp. 145–160.

[33] C. A. Waldspurger, "Lottery and stride scheduling: Flexible proportional-share resource management," in *Ph.D. dissertation, Massachusetts Institute of Technology*, September 1995.

[34] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. of the 41st Intl. Symp. on Microarchitecture*, 2008, pp. 318–329.

[35] X. Wang and J. Martinez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *High Performance Computer Architecture (HPCA'15), 21st Intl. Symp. on*, 2015, pp. 113–125.

[36] S. M. Zahedi and B. C. Lee, "REF: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proc. of the 19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 145–160.

[37] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: An approach for extending smps," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX, 2000, pp. 25–36.

[38] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of USENIX Annual Technical Conference*, 2005.

[39] A. Hansson, N. Agarwal, A. Kolli, T. F. Wenisch, and A. N. Udipi, "Simulating DRAM controllers for future system architecture exploration," in *2014 Intl. Symp. on Performance Analysis of Systems and Software*, 2014, pp. 201–210.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[41] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 45–57.

[42] S. Eyerman and L. Eeckhout, "Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 93–96, July 2014.

[43] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, pp. 5–, Aug. 2004.

[44] A. Limited, *ARMv8 Architecture Reference Manual*, 2016.

[45] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das, "METE: Meeting end-to-end QoS in multicores through system-wide resource management," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 13–24, Jun. 2011.

[46] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *Proceedings of the 7th Conference on File and Storage Technologies*, ser. FAST '09, 2009, pp. 85–98.

[47] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, pp. 374–385.

[48] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, ser. SIGCOMM '94, 1994, pp. 24–35.

[49] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.

[50] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-based congestion control for the datacenter," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15, 2015, pp. 537–550.