Generalizing QoS-Aware Memory Bandwidth Allocation to Multi-Socket Cloud Servers

David Gureya*†, João Barreto*, and Vladimir Vlassov†
*INESC-ID, Universidade de Lisboa, Lisbon, Portugal
Email: {david.gureya,joao.barreto}@tecnico.ulisboa.pt
† KTH Royal Institute of Technology, Stockholm, Sweden
Email: {daharewa,vladv}@kth.se

***Abstract*—Although the problem of QoS-aware resource allocation is not new, novel hardware-based resource allocation mechanisms have recently become available in commodity cloud servers and enabled a new generation of QoS-aware resource allocation approaches. Unfortunately, to the best of our knowledge, existing proposals are by design tailored to single-socket architectures only. In many warehouse scale data centers, dual-socket (or even larger) machines already constitute the largest share of hosts. This paper presents the full design and implementation of BALM, a QoS-aware memory bandwidth allocation technique for multi-socket architectures. BALM combines commodity bandwidth allocation mechanisms originally designed for single-socket with a novel adaptive cross-socket page migration scheme. Our evaluation with a large and dynamic set of real applications co-located on a dual-socket machine shows that BALM can overcome the efficiency limitations of state-of-the-art. BALM delivers substantial throughput gains to bandwidth-intensive best-effort applications, while ensuring marginal SLO violation windows to latency-critical applications.**

***Index Terms*—QoS-aware resource allocation, Cloud computing, Multi-socket systems**

I. Introduction

One prominent way to reduce infrastructural costs in the Cloud is through *workload consolidation*, i.e., by co-locating applications on the same physical host. Among the co-located applications, some have quality of service (QoS) requirements, as determined by one or more service-level objectives (SLOs), and are commonly called *latency-critical* applications (LCAs). In contrast, the so-called *best-effort* applications (BEAs) have no SLO and are simply meant to run in background in a throughput-oriented fashion.

The co-located applications contend for shared resources, such as network and storage bandwidth, CPU cores, last-level caches (LLC), and memory. If allowed to run in the wild, the co-located system can easily incur *noisy neighbour* phenomena, in which the resource demands of some applications degrades the performance of other co-located applications up to a point where certain LCAs start violating their SLOs. Therefore, consolidating LCAs and BEAs in the same host poses a challenging *QoS-aware resource allocation problem*: the shared resources should be allocated in such a way that safeguards the SLO of LCAs while maximizing the throughput of the BEAs. This problem is dynamic by nature, as running applications may have distinct phases with different resource usage patterns, while active applications leave upon completion, and new ones may join at any time. Therefore, appropriate solutions should react to such changes by efficiently reallocating resources while ensuring that SLOs are violated only for negligible periods.

Although this problem is not new, novel hardware-based resource partitioning mechanisms have recently become available in commodity cloud servers and enabled a new generation of QoS-aware resource allocation approaches. One notable example is the support for hardware-based partitioning of LLC and memory bandwidth as provided by Intel Resource Director Technology (RDT) [1]. Recent proposals such as PARTIES [6] and CLITE [37] exploit such new mechanisms to enforce QoS-aware resource allocation with unprecedented effectiveness.

Another significant technological trend is the growing prevalence of multi-socket systems in the cloud. In many warehouse-scale data centers, dual-socket (or even larger) machines already constitute the largest share of hosts [22]. Unfortunately, PARTIES and CLITE, as well as their predecessors, are, by design, tailored to a single socket.

Although deploying such proposals directly in multi-socket hosts is possible, it incurs important limitations [30]. One notable limitation is that they prevent an application running to place data pages on remote memory nodes. This essentially *disallows cross-socket sharing of memory*, which entails a sub-optimal use of multi-socket host's aggregate memory resources. This issue is especially relevant given the increased prevalence of memory-intensive applications [21], [24], [36]. As an example, consider the case where a memory-intensive BEA, A, runs in one socket and saturates the local memory bandwidth, while a CPU-intensive LCA, B, runs on another socket and only places a negligible access demand on the local memory. Allowing A to place a portion of its pages in the idle remote memory node would boost A by providing it with an improved (aggregate) memory bandwidth, while not causing harmful interference with B.

Therefore, in order to properly utilize over-provisioned memory resources in multi-socket hosts, state-of-the-art QoS-aware resource allocation systems need to be generalized

This work was partially supported by Fundação Ciência e Tecnologia (FCT) under grant UIDB /50021/2020, and the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under FPA 2012-0030.

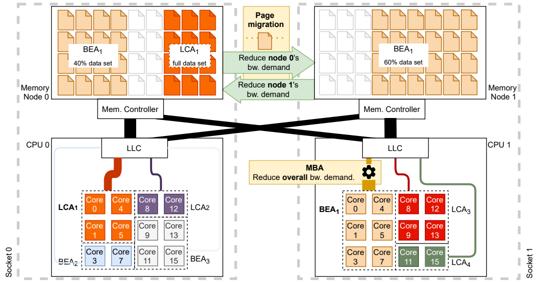


Fig. 1: QoS-aware bandwidth allocation in a dual-socket across multiple BEAs and LCAs.

to allow cross-socket sharing of memory as in the previous example. This paper addresses the above goal.

As a **first contribution**, we present the full design and implementation of BALM (memory Bandwidth ALlocation for Multi-socket), a QoS-aware memory bandwidth allocation technique for cross-socket sharing of memory in multi-socket architectures. The key insight is to combine commodity bandwidth allocation mechanisms originally designed for single-socket – namely, Intel RDT's memory bandwidth allocation (MBA) mechanism – with a novel adaptive cross-socket page migration scheme. By doing so, BALM overcomes the efficiency limitations of the original mechanisms when deployed in multi-socket scenarios. BALM relies on this novel approach to allow multiple LCAs and BEAs to run together in the same multi-socket host while sharing over-provisioned memory resources.

As a **second contribution**, we evaluate BALM by co-locating, in a dual-socket system, real LCAs (namely, the Memcached key-value store [29] and the Xapian probabilistic information retrieval library [4]) with realistic memory-intensive BEAs. Our evaluation shows that BALM can safeguard the LCAs with marginal SLO violation windows, while delivering up to 87% throughput gains to bandwidth-intensive BEAs when compared to state-of-the-art alternatives.

The contributions in this paper extend a preliminary paper [30] that studies the limitations of existing allocation mechanisms and presents a simplistic outline of BALM's approach.

The rest of the paper is organized as follows. Section II states the problem and system model. Section III presents BALM. Section IV evaluates BALM against state-of-the-art alternatives in realistic co-location scenarios. Section VI draws conclusions and presents future work.

II. Problem statement

Recent papers [6], [17], [20], [37] formulate the problem of QoS-aware resource allocation problem as follows. In a given server, multiple LCAs and BEAs run together. The LCAs are governed by an SLO. The SLO should be guaranteed most of the time (e.g., 99% of time). In contrast, BEAs have no SLO. These applications run in background, utilizing any spare resources (left by the LCAs) according to some best-effort policy to maximize the BEA's throughput.

This paper aims at generalizing QoS-aware resource allocation to workload consolidation in multi-socket servers, with a specific emphasis on cross-socket memory bandwidth allocation. Hence, we need to complement the previous problem definition with additional restrictions to embrace the additional complexity of multi-socket workload consolidation scenarios such as the one that Figure 1 illustrates.

In a multi-socket system, each socket comprises multiple multi-core CPUs and memory nodes. For presentation simplicity, and without loss of generality, we assume that each socket only holds a single CPU and a single memory node. The threads running at a given CPU can both access the local and remote memory nodes. Hence, the different memory nodes form a non-uniform memory access (NUMA) architecture. We assume that some application placement system (e.g., [14], [33]), selects which applications run on a given host/socket. We also assume the common setting where the threads of any given application all run on the same socket of the host.

Among the shared resources,we restrict our focus to the allocation of memory bandwidth. Therefore, we assume that the applications may only interfere through memory bandwidth contention, while contention on other kinds of resources is negligible or has been taken care of by some other means.

We assume the pages of an LCA are exclusively mapped to the local memory node. In contrast, BEAs are allowed to place their pages across multiple memory nodes, to benefit from the spare memory bandwidth. For an important class of BEAs, memory bandwidth, rather than access latency, is the main bottleneck. It is well studied that, for such bandwidth-intensive applications, interleaving its pages across the available nodes (both local and remote) can maximize throughput since it provides its threads with a higher aggregate bandwidth [33], ideally with larger fractions of pages in the memory nodes that offer higher bandwidth [15]. We further assume that the LCAs running on some socket do not saturate the bandwidth of the local memory node by themselves. Consequently, any SLO violation on a given socket can always be fixed by reducing, to some extent, the memory demand placed by the BEAs on that socket's local memory.

Hereafter, when an LCA does not meet its SLO, we say that the system is in an *invalid configuration*. Otherwise, the system is said to be in a *valid configuration*. Whenever the system enters an invalid configuration, we can employ some bandwidth allocation mechanism to transition to a valid configuration again (i.e., fix the SLO violation(s)). As formulated in the previous section, that transition should ideally: i) move to a valid configuration as soon as possible; and ii) reach a configuration that, among the available valid configurations, maximizes the throughput of the BEA.

In a previous study, we have shown that existing mechanisms for memory bandwidth allocation exhibit important shortcomings when employed to implement QoS-aware memory bandwidth allocation in multi-socket architectures [30]. For instance, Intel RDT's memory bandwidth allocation (MBA) mechanism imposes a relevant cost on the performance of the BEAs. The reason is that, in order to heal an SLO violation happening on *a specific* socket (where one or more victim LCAs are running), using MBA to throttle down the

Algorithm 1: BALM's main control loop

```
1   begin
2       retries = 0;
3       while true do
4           {color, violationLocation} = evaluateSLO(LCAs);
5           if color ∈ {yellow, red} then
                /* We need to adapt BEAs (strongest contributors first)   */
6               orderedBEAs = orderByMemUsage(SLOlocation);
                /* 1. Try to fix violation(s) ASAP with aggressive MBA    */
7               for each BEA in orderedBEAs do
8                   color = setMBA(BEA, minMBA);
9                   if color ∈ {grey, green} then
10                      break;
11              for each BEA in orderedBEAs do
12                  if —SLOlocation— == 1 then
13                      if SLOlocation = BEA.socket then dir=outbound;
14                      else dir=inbound;
15                  else
16                      dir = none;
                    /* 2. Adapt BEAs to find better valid configuration  */
17                  while (BEA.MBA< 100) do
18                      if color ∈ {grey, green} then
                            /* 2a. There is room to alleviate MBA         */
19                          color = incMBA(BEA);
20                      if color ∈ {yellow, red} then
                            /* 2b. Try to fix violation by migrating
                               enough pages away from victim              */
21                          color = migratePagesUntilGrey(BEA, dir);
22                          if color ∈ {yellow, red} then
23                              break;
24              if color == red then
25                  if retries>MAXRETRIES then  throw CannotFixSLOViolationException ;
26                  else retries++;
27              else retries=0;
28          else if color == green then
29              retries = 0;
30              if BEAs = BELoadChanged() then
31                  optimizeConfig(BEAs);
```

noisy neighbor BEA(s) will unnecessarily slow down the memory accesses of the BEA(s) on *every* memory node.

As our study has shown [30], migrating pages of the noisy neighbour BEA(s) away from the memory node where the victim LCA runs can be a more efficient alternative to MBA (or other single-socket mechanisms). However, on the downside, page migration has substantial costs. Not only it requires intensive data movement across different memory nodes, but it also has well known expensive management overheads – most notably, kernel memory management and synchronization [28]. For this reason, page migration is unsuitable to QoS-aware memory bandwidth allocation, if used as a stand-alone mechanism.

III. BALM

This section presents BALM, a QoS-aware memory bandwidth allocation mechanism for multi-socket hosts. BALM combines MBA and page migration in an unprecedented way that eliminates each mechanism's shortcomings while delivering the best of both worlds. Section III-A describes the algorithm, and Section III-B addresses implementation details.

While the general approach of BALM is easily generalized to multi-socket systems of large sizes, this paper focuses on dual-socket systems only. We leave the evaluation of BALM in larger systems to future work.

A. Algorithm

The architecture of BALM comprises a memory bandwidth allocation component and a monitoring component. The former controls the MBA and page migration mechanisms to allocate the available memory bandwidth of the multi-socket host to BEAs. The latter continuously collects LCA's SLO metrics (e.g., tail latency) to detect violations and throughput of the BEAs. Its outcome can be: *red*, which means that at least one LCA is in an invalid configuration, i.e., at least one SLO violation is occurring; *yellow*, which means that, although the system is in a valid configuration, at least one SLO violation is likely to occur soon, since at least one LCA's SLO metric is less than thr_{yellow} below the SLO target; *green*, when the system is in a resourcefully valid state, since every LCA meets its SLO target by a large enough margin (as defined by thr_{green}); and grey otherwise (between yellow and green). Upon yellow or red states, the monitoring component also indicates in which sockets reside the LCAs that are prone to or experiencing SLO violations (resp.).

At the heart of BALM lies the controller, which dynamically adjusts memory bandwidth allocations between consolidated applications using fine-grained monitoring and memory bandwidth partitioning, to satisfy LCA's QoS and maximize BEA's performance. For presentation brevity, in this section, we refer to the controller as simply BALM. BALM reacts to input fed by the monitoring component by triggering actions in the memory bandwidth allocation component. Its design materializes the design guidelines that our preliminary paper laid down [30]. Algorithm 1 summarizes the decision making flow of BALM. Periodically, it reads the latest system-wide SLO evaluation provided by the monitoring component. Two very distinct actions may arise depending on such evaluation.

Healing SLO violations. BALM's most critical action occurs when it learns that an SLO violation is happening (red) or prone to occur (yellow). In this case, the BEAs are ordered in decreasing order of the memory bandwidth recently consumed from the problematic memory node(s) (line 6).

BALM handles yellow or red situations in two main phases. The first phase aims at quickly preventing or fixing (resp.) the SLO violation(s) by aggressively enabling MBA at its most restrictive level (MBA 10) to each BEA in the ordered list until the system moves to a green state (lines 7-10).

The second phase then takes place, which attempts to maximize the throughput of the BEAs affected by the first phase, one by one in the same order, by adapting the memory bandwidth allocated to them. For each such application, the second phase executes a 2-dimensional hill-climbing, combining MBA steps and page migration steps. Upon each step, the system SLOs are evaluated again, since this outcome determines the next step to take in the online search. Therefore, the second phase typically takes much more time than the first one. In a best-case scenario, when this phase completes, every BEA will be running with no MBA restrictions (MBA 100), at the local-to-remote page ratio that optimizes its throughput while ensuring a green state. However, as we detail next, that might not always be possible.

Each iteration starts by checking if the system is in a green state. If so, then probably there is enough spare memory bandwidth to alleviate the current MBA restriction (i.e., increase MBA level) without raising a new SLO violation (lines 17-19). In contrast, the second step is taken only when the system is in a danger state (yellow or red). This step consists of migrating enough pages of the BEA *away* from the socket where the SLO violation is prone to/already happening until the system returns to a green state (lines 20-21). Pages are migrated using the weighted interleaving migration technique of BWAP [15], complemented with an SLO validation that, upon migrating a fraction of pages in the desired direction, checks whether the system has entered a green state (and returns) or not (migrates an additional fraction, if available).

A BEA can take multiple iterations to converge to the ideal configuration. However, each iteration does not necessarily run both steps. In some worst-case scenarios, BALM will not be able to find a valid configuration where every BEAs run MBA-free. A first, obvious case is when there exists no valid configuration. When BALM suspects it is in such a situation, it throws an exception, which is expected to be handled by some higher layer that will solve the problem through stronger measures – such as migrating some applications to another host in the data center. A second worst-case scenario is when both sockets simultaneously suffer from SLO violations (either happening or prone to). In this case, the migration step is skipped (i.e., function *migratePagesUntilGreen* does nothing), since there is no chance of migrating pages in either way (from an invalid configuration). Consequently, the SLO violation can only be fixed/prevented by resorting to MBA.

Thirdly, we note that the algorithm that handles SLO violations assumes that the system remains in a steady state – regarding the set of deployed applications and their load – until the BEAs have all been adapted to the final valid configuration. If a significant change to that steady state occurs, it is easy to show that the algorithm might no longer converge to an appropriate configuration. In the worst extreme, a sudden disruption in the middle of the algorithm may push it towards an invalid configuration. In this case, BALM will repeat the whole procedure, up to a given number of retries (line 25). Meanwhile, if the system stabilizes, BALM will finally reach the desired (valid and optimized) configuration.

Re-configuring upon workload changes. When every LCA meets its SLO target by a safe margin (i.e., system state is green), some BEA pages may be moved back to the remote node to optimize its performance (line 31). This allows the excess memory bandwidth to be reclaimed, improving overall system utilization.

As a final note, we highlight the importance of appropriately setting the thr_{yellow} and thr_{green} thresholds. Larger values of thr_{yellow} make BALM more proactive at detecting imminent SLO violations; however, they also render BALM susceptible to false alarms, which hurt resource efficiency. Larger values of thr_{green} may lead to low resource utilization, while smaller values increase the risk of BALM choosing under-provisioned configurations which quickly lead to new SLO violations.

B. Implementation

We have implemented BALM as a user-level controller that polls the SLO metric and memory bandwidth utilization of applications and interacts with the OS (Linux) and hardware to adjust memory bandwidth allocations. The controller is pinned on core 0, taking at most 10% of its core utilization.

SLO monitoring. We rely on the existence of per-application monitoring plug-ins, which can reside at either the client or server sides. Each such plug-in monitors the SLO metric of a given LCA. For instance, with Memcached, we use a client-side component that measures the tail latency of requests (more details in Section IV).

Page migration mechanism. The page migration mechanism of BALM uses the approximated online page placement open source tool proposed in BWAP [15]. We complement BWAP with an SLO validation component that checks whether the system is in violation or not when a fraction of pages is migrated in the desired direction.

MBA mechanism. We use an interface provided by Intel to throttle MBA dynamically at runtime [3]. MBA is a per-core mechanism. However, for efficiency, BALM does not tune MBA on a per-thread granularity. Instead, BALM tags all the threads of the same BEA with a unique *class of service* (CLOS), when the BEA starts. When BALM needs to apply a new MBA level to the BEA, BALM simply sets the MBA level of the corresponding CLOS. This implicitly throttles all threads of the BEA by the new MBA level.

IV. Evaluation

Our evaluation addresses two key questions: *1. What performance advantage does* BALM *bring to memory-intensive BEAs on dual-socket NUMA systems? 2. How effective is* BALM *in fixing SLO violations?*

A. Experimental methodology

To answer each question, we study how BALM and other state-of-the-art alternatives handle QoS-aware memory bandwidth allocation in a complex and dynamic workload consolidation scenario. We use the execution time of BEAs and SLO violation time of LCAs as the performance metrics that provide quantified answers to each question, respectively. Every experiment is repeated 5 times, so the results presented in this section are average of such runs.

Dual-socket machine. We evaluate BALM on a dual-socket machine with two Intel Xeon Gold 5218 CPUs, with 16 cores per CPU, 64GB DRAM (32GB at each NUMA node), running Linux 4.19. It supports MBA, with 8 available levels.

LCA workloads. As representative LCAs, we consider Memcached [29] and Xapian [4] in our experiments.

Memcached is a widely-adopted, high-performance, distributed object caching system that is mainly used to speed up web requests by caching data and objects in memory and is a critical tier in many cloud services [12]. We use Memcached 1.5.22, compiled from its official source. For each LCA instance in the evaluated scenarios, we run Memcached with the default/recommended number of threads, i.e., 4 threads pinned to 4 physical cores. We also assign 8 cores to handle network interrupts (IRQ). It is well studied that allowing application threads to share cores with IRQ handlers leads to lower throughput and higher latency [6], [25]. Except where stated, our default Memcached deployment is 10 million items, each with a 30B key and a 200B value; the SLO target is set

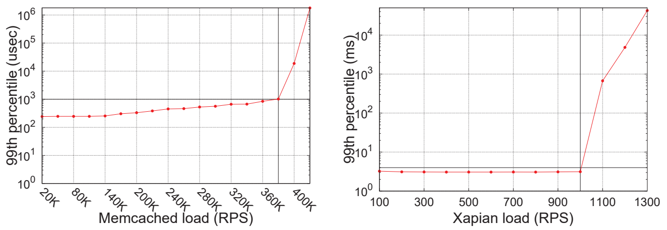


Fig. 2: Tail latency with increasing load of Memcached and Xapian. The vertical line shows the *max load*, while the horizontal line shows the target SLO. The y-axis is logarithmic.

to 1ms for 99^{th} percentile latency, which is in line with the experimental deployment methodology in previous works [6], [25], [37], [38].

We use an in-house, open-loop workload generator, similar to *Mutilate* [2], as Memcached client. Clients run on machines on the same network as dual-socket machine, where Memcached runs. The load generator uses exponential inter-arrival time distribution, similar to the query distributions at Facebook [6], [12], [34]. We also limit input loads to read-only, which corresponds to the majority of requests in production systems, e.g., 95% of Memcached requests at Facebook [6], [12].

Xapian is an open-source search engine from the Tailbench suite [31]. The search index is built from a snapshot of the English version of Wikipedia. We use the default configuration and open-loop load generators provided by Tailbench [31]. The load generator chooses the query terms randomly, following a Zipfian distribution. This has been shown to model online search query distributions well [31]. The SLO target is set to 5ms for 99^{th} percentile latency.

We assume that the SLO of the LCAs is defined by tail latency (99^{th} percentile) of request-to-response latency, as observed on their clients' sides. Similarly to other works on QoS-aware resource allocation [6], [37], [38], we first study the impact of increasing input load on the tail latency of each LCA to determine estimate reasonable targets for its SLO and quantify the maximum achievable throughput that our platform can sustain. We run each LCA in isolation, starting from a low load (requests per second, RPS) and gradually increase the load until it starts dropping requests on the server-side. Figure 2 shows the relationship between tail latency and input load (RPS) for each LCA. Both LCAs exhibit a rapid increase in tail latency after exceeding a certain load threshold. We set the target SLO as the 99^{th} percentile latency of the curve's knee, as indicated by the horizontal line in Figure 2. Consequently, the RPS at the knee of the curve is denoted as the *max load*, which is the maximum throughput that the platform can sustain without violating SLO in an interference-free system.

To monitor the SLO of the LCAs, BALM's monitoring component keeps a sliding window of all the recent requests that have occurred in the last n seconds and polls the SLO metric, such as tail latency at m milliseconds interval (which is fine-grained). We configure BALM with n and m to 3 seconds and 20 ms, respectively. This choice of parameters allowed the SLO metric to be calculated over large-enough samples,

Benchmark	Bw. requirements (GB/s		Description
	Reads	Writes	
MG.C (MG)	26.31	7.16	Multi-Grid on a sequence of meshes
Ocean_cp (OC)	23.81	8.48	Simulates large-scale ocean movements
SP.C (SP)	20.48	10.76	Scalar Penta-diagonal solver
UA.C (UA)	16.79	5.40	Unstructured Adaptive mesh, dynamic and irregular memory access
Blackscholes (BS)	2.50	0.35	Option pricing with Black-Scholes Partial Diff. Equation (PDE)
EP.B (EP)	0.01	0.01	Embarrassingly Parallel
Swaptions (SW)	0.01	0.01	Financial analysis

TABLE I: Evaluated BE benchmarks

which reduce measurement noise; while allowing BALM to react quickly after a sample yields an SLO violation.

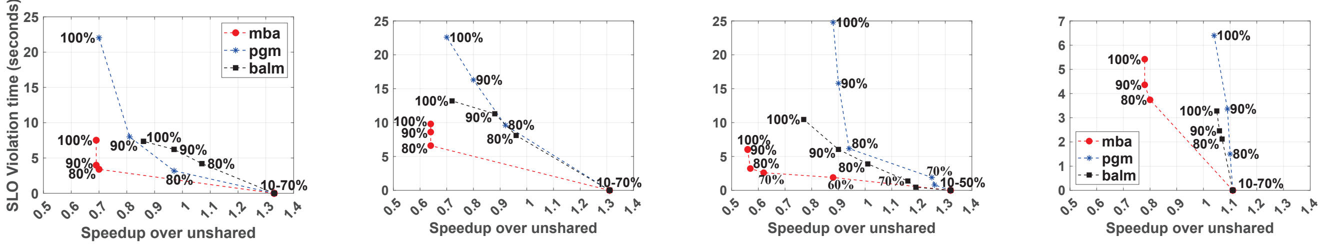
Further, we set the threshold parameters of BALM that trigger the yellow and green states (thr_{yellow} and thr_{green}, resp.) discussed in section III-A to 5% and 20% below the target SLO metric (resp.). We chose these two thresholds based on a sensitivity analysis on a subset of examined applications. Then, we used those values for every other application/experiment.

BEA workloads. For the bandwidth-intensive BEAs, we used memory-intensive benchmarks from several benchmark suites, i.e., NAS [16], PARSEC [5] and SPLASH [26]. Table I lists all the benchmarks used for our evaluation. These benchmarks represent a wide diversity of application domains which are typically throughput-oriented, which are also used as such in related QoS-aware resource allocation works (e.g., [20], [37]). The selection criterion was as follows: we measured each benchmark's memory traffic and selected the benchmarks that incur higher memory traffic when allocated with a single socket's full resources. All the evaluated benchmarks are multi-threaded. We pin the threads of each benchmark on the cores allocated to it. All BEAs are characterized by multiple phases with different memory intensities.

Alternative solutions. We compare BALM to *MBA* (*mba*) and *page migration (pgm)*, each used stand-alone; as well as an *unshared* approach, in which we disallow any cross-socket memory bandwidth sharing by imposing that the bandwidth-intensive BEA only places pages on its local socket.

We follow up our preliminary work [30], which had evaluated BALM in a simplified small-scale scenario. In contrast, in this paper we consolidate six applications. This provides a reasonably large scale scenario, with many applications to monitor and manage in an inherently dynamic environment – with frequent workload changes, not only due to applications starting/ending, but also due to phase changes within each application. The application mix comprises: Memcached and Xapian (LCAs); BS, EP and SW (BEAs with low to moderate memory intensity); and one bandwidth-intensive BEA (either OC, MG, SP, or UA, each one selected in a distinct experiment). Therefore, two applications stand out for their bandwidth demand: Memcached and the bandwidth-intensive BEA. Hence, although the six applications need to be monitored and managed, the main challenge is to address any harmful interference arising between the latter two applications.

Despite the many possible application-to-socket combinations, what essentially distinguishes all of them is whether the two bandwidth-intensive applications reside at the same socket or on opposite sockets. For space limitations we only



(a) OC (b) MG (c) SP (d) UA

Fig. 3: Performance of BEAs and SLO violation time of high-load LCA. The plots show the speedup of BEAs (x-axis) and SLO violation time of LCA (y-axis) that can be achieved by different mechanisms when LCA is running at the fraction of its *max load* indicated by the % values.

show results for the same-socket scenario. We note that our experiments with both applications on opposite sockets yielded very similar conclusions as with same-socket scenario.

For each experiment, one bandwidth-intensive BEA is chosen (OC, MG, SP, or UA). Both LCAs (Memcached and Xapian) run for the whole experiment. The load of Xapian is fixed at 100% for the whole experiment, while the load of Memcached varies, in phases, from 10% to 100%. At the beginning of each Memcached phase, we simultaneously launch all 4 BEAs – the chosen bandwidth-intensive BEA, and BS, EP and SW). Each Memcached phase ends as soon as every BEA has completed (note that different BEAs execute for different periods), then the next phase starts.

B. Results

Figure 3 presents the results for each metric (BEA performance and LCA SLO violation time) for increasing LCA loads. As expected, when the LCA runs at a modest load levels, no SLO violation occurs and the BEA achieves its maximum performance since it runs with no bandwidth allocation restrictions – regardless of which mechanism is used. This corresponds to the bottom-right point at each plot in Figure 3.

However, as we increase the LCA load beyond a critical level (which, depending on the bandwidth intensity of each BEA, ranges between 70% and 90% of *max load*), QoS violations arise at increasing frequency and intensity. These trigger the different mechanisms to allocate less memory bandwidth to the BEA, thus reducing its throughput. Figure 3 also makes it evident that, in such high load situations, each mechanism handles the SLO violations with very distinct effectiveness. As one increases the LCA load beyond a critical level, the *mba* curve quickly expands towards the left-hand extreme of the plot (i.e., sacrifices the throughput of the offending BEA), while *pgm* quickly grows upwards (i.e., taking an increasingly longer time to heal SLO violations).

In contrast, BALM's curves in the same plots manage to stay closer to the initial optimal point (the low-load point). Hence, BALM handles increasing LCA loads at relatively lower costs on *both* axis. Most importantly, if we chose a given LCA load and observe how each mechanism performs at both criteria, then it becomes clear that BALM's performance on each axis is typically close to the alternative mechanism that is best-performing in that axis. More precisely, BALM is able to outperform *mba* and *unshared* by up to $1.87\times$ and $1.33\times$, resp.. To understand why BALM does not always achieve the same BEA throughput as *pgm*, recall that BALM activates MBA until the page migration process completes, which temporarily hinders the BEA.

Finally, we observed that, in situations of extreme memory bandwidth interference, only using *mba* is insufficient to fix SLO violations, therefore the SLO violation can last until (at least) one of the conflicting applications switches to a lower-load phase. Contrary, BALM 's more aggressive combination of *mba* and *pgm* is able to fix the SLO violation even before the workloads change. Figure 3 (d) is an example of the above situation. The above results confirm that BALM attains the virtues of each extreme (*mba* and *pgm*), making BALM a well-balanced compromise between both conflicting criteria.

V. Related Work

Architectural and system software techniques to tackle interference in a multi-tenant environment have been extensively explored. These techniques can be grouped into three broad approaches. The first is to simply avoid sharing resources with LCAs [11], [17], [18], [23]. This approach preserves the QoS of the LCAs but lowers the resource efficiency of the system. The second approach avoids co-scheduling of applications that may interfere with each other [8]–[10], [13], [35]. Although this approach improves resource utilization, it limits application co-scheduling options and requires some offline/prior knowledge of the co-scheduled applications. Finally, interference can be eliminated by partitioning resources among consolidated applications using OS- and hardware-level isolation techniques [6], [7], [17]–[19], [32], [36], [37]. This approach has the following benefits: (1) it maximizes resource utilization and throughput, or trades off throughput vs. fairness [27], [36]; (2) it provides QoS for LCAs [6], [17], [37]. BALM's approach falls under this approach. BALM implements a robust policy that guarantees QoS by effectively employing OS-level page migration and hardware-level MBA mechanisms.

More recent proposals [6], [20], [37] have focused on the QoS-aware resource allocation problem that is the departing point to our reformulation in Section II, where LCAs are consolidated with BEAs, to safeguard the SLO of LCAs while maximizing the throughput of BEAs. However, to the best of our knowledge, existing proposals to that problem, have consolidated applications run in a single socket system. This recent research has inspired our proposal. We differentiate from these works, as our focus is on multi-socket servers.

Memory bandwidth partitioning has been recently used to enhance performance and fairness. EMBA [27] introduced a performance model to guide the use of MBA to improve performance. CoPart [36] proposed a resource manager that uses Intel RDT to dynamically partition the LLC and memory bandwidth to the applications. Still, these approaches are designed for single-socket servers. Further, they treat applications as of equal priority, thus lack support for QoS.

The most relevant works to BALM are BWAP [15], PARTIES [6], Heracles [17] and CLITE [37]. PARTIES, CLITE, and Heracles rely on resource partitioning to guarantee cross-application isolation. However, these systems are designed for single-socket servers. Moreover, both PARTIES and Heracles do not exploit hardware support for memory bandwidth partitioning. The lack of hardware support for memory bandwidth isolation complicates and constrains the efficiency of any system that dynamically manages workload consolidation [17]. A recent alternative has been proposed in BWAP [15], which enables cross-socket memory bandwidth allocation among two or more applications running in disjoint sockets. However, BWAP does not support dynamic scenarios, where the overall system behavior may change over time. Most importantly, BWAP lacks support for QoS and does not exploit MBA.

VI. Conclusions

This paper presents the full design and implementation of BALM, a QoS-aware memory bandwidth allocation technique for multi-socket architectures in the cloud. By combining commodity bandwidth allocation mechanisms originally designed for single-socket with a novel adaptive cross-socket page migration scheme, BALM can overcome the efficiency limitations of today's state-of-the-art when deployed in multi-socket scenarios. Our large-scale experimental evaluation with real applications on a dual-socket machine shows that BALM can ensure marginal SLO violation windows while delivering up to 87% throughput gains to bandwidth-intensive best-effort applications, when compared to state-of-the-art alternatives.

This work leaves two main open questions to be addressed in future work. First, the novel memory bandwidth allocation of BALM can be integrated into more larger frameworks such as CLITE [37] or PARTIES [6], in order to generalise them to multi-socket scenarios, while supporting QoS-aware allocation of other kinds of resources that are not handled by BALM. Second, while BALM is designed and implemented to support systems with a larger and more complex socket topology, the effectiveness of BALM still needs to be experimentally evaluated in such settings.

References

[1] Intel® resource director technology reference manual, 2019.

[2] Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate. Accessed: 2020-06-02.

[3] User space software for intel(r) resource director technology. https://github.com/intel/intel-cmt-cat. Accessed: 2020-10-05.

[4] Xapian project website. https://github.com/xapian/xapian. Accessed: 2021-02-08.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] S. Chen, C. Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. ASPLOS '19.

[7] C. Delimitrou and C. Kozyrakis. Bolt: I know what you did last summer... in the cloud. ASPLOS '17.

[8] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. ASPLOS '13.

[9] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. ASPLOS '14.

[10] C. Delimitrou and C. Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. *SIGARCH Comput. Archit. News*, 44(2):473–488, March 2016.

[11] A. Verma et al. Large-scale cluster management at google with borg. EuroSys '15.

[12] B. Atikoglu et al. Workload analysis of a large-scale key-value store. SIGMETRICS '12.

[13] C. Delimitrou et al. Tarcil: Reconciling scheduling speed and quality in large shared clusters. SoCC '15.

[14] D. Goodman et al. Pandia: Comprehensive contention-sensitive thread placement. EuroSys '17.

[15] D. Gureya et al. Bandwidth-aware page placement in numa. IPDPS '20.

[16] D. H. Bailey et al. The nas parallel benchmarks–summary and preliminary results. Supercomputing '91.

[17] D. Lo et al. Heracles: Improving resource efficiency at scale. ISCA '15.

[18] David Lo et al. Towards energy proportionality for large-scale latency-critical workloads. ISCA '14.

[19] H. Kasture et al. Rubik: Fast analytical power management for latency-critical systems. MICRO-48.

[20] J. Fried et al. Caladan: Mitigating interference at microsecond timescales. OSDI '20.

[21] L. Tang et al. The impact of memory subsystem resource sharing on datacenter applications. ISCA'11.

[22] L. Tang et al. Optimizing google's warehouse scale computers: The numa experience. HPCA '13.

[23] M. Schwarzkopf et al. Omega: Flexible, scalable schedulers for large compute clusters. EuroSys '13.

[24] S. Blagodurov et al. A case for numa-aware contention management on multicore systems. USENIXATC'11.

[25] S. Chen et al. Workload characterization of interactive cloud services on big and small server platforms. IISWC '17.

[26] S. W. Woo et al. The splash-2 programs: Characterization and methodological considerations. ISCA '95.

[27] Yaocheng Xiang et al. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. ICPP 2019.

[28] Zi Yan et al. Nimble page management for tiered memory systems. ASPLOS '19.

[29] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, August 2004.

[30] D. Gureya, V. Vlassov, and J. Barreto. Balm: Qos-aware memory bandwidth partitioning for multi-socket cloud nodes. SPAA '21.

[31] H. Kasture and D. Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. IISWC '16.

[32] H. Kasture and D. Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. ASPLOS '14.

[33] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. USENIX ATC '15.

[34] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. EuroSys '14.

[35] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. ISCA '13.

[36] J. Park, S. Park, and W. Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. EuroSys '19.

[37] T. Patel and D. Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. HPCA '20.

[38] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *SIGOPS Oper. Syst. Rev.*, 2016.