Doctoral Thesis in Information and Communication Technology

Resource Allocation for Data-Intensive Services in the Cloud

DAVID DAHAREWA GUREYA

Stockholm, Sweden 2021

TÉCNICO
LISBOA

Resource Allocation for Data-Intensive Services in the Cloud

DAVID DAHAREWA GUREYA

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Friday the 19th of November 2021 at 14:00 in Sal B, Kistagången 16, Kista.

Doctoral Thesis in Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2021

ISBN 978-91-8040-045-9
TRITA-EECS-AVL-2021:70

Printed by: Universitetsservice US-AB, Sweden 2021

To my family. Thanks for Everything.

Abstract

Cloud computing has become ubiquitous due to its resource flexibility and cost efficiency. Resource flexibility allows Cloud users to elastically scale their Cloud resources, for instance, by horizontally scaling the number of virtual machines allocated to each application as the application demands change. However, matching resource demands to applications is non-trivial and applications experiencing highly dynamic workloads make it much more difficult. Cost efficiency is primarily achieved through workload consolidation, i.e., by co-locating applications on the same physical host. Unfortunately, workload consolidation often comes at a performance penalty, as consolidated applications contend for shared resources, leading to interference and performance unpredictability. Interference is particularly destructive for latency-critical applications, which must meet strict quality of service (QoS) requirements. Another significant technological trend is the growing prevalence of multi-socket systems in contemporary data centers. However, to the best of our knowledge, existing proposals for QoS-aware resource allocation are, by design, not tailored to multi-socket systems. Specifically, existing proposals do not support cross-socket sharing of memory, which entails a sub-optimal use of multi-socket host's aggregate memory resources.

This thesis focuses on two aspects of Cloud resource management namely, QoS-aware elasticity and resource arbitration, on two levels: inter-node resource management and intra-node resource management. In the first level, we consider the number of virtual machines (VMs) as the main resource to allocate and de-allocate for horizontal auto-scaling of an elastic service or application in the Cloud. In the intra-node resource management, we treat the memory bandwidth in multi-socket system as the resource to arbitrate among co-located applications. In both levels, the overall goal of this thesis is to provide resource management mechanisms that automatically adapt the resources allocated to data-intensive services to improve resource utilization while meeting service-level objectives (SLOs).

In the context of inter-node resource management for auto-scaling of elastic Cloud services, this thesis improves the usefulness of elasticity controllers by addressing some of the challenges posed by current model-predictive control systems (such as training and tuning of the controller and adapting it to different workload patterns). To enable elastic execution of Cloud-based services using model-predictive control, we propose, implement, and evaluate OnlineElastMan, a self-trained proactive elasticity manager for Cloud-based storage services. OnlineElastMan excels its peers with its practical aspects, including easily measurable and obtainable performance and QoS metrics, automatic online train-

ing, and an embedded generic workload prediction module. Our evaluation shows that OnlineElastMan continuously improves its provision accuracy, minimizing provisioning cost and SLO violations, under various workload patterns.

In the context of intra-node resource management, this thesis departs from the observation that, since state-of-the-art QoS-aware resource allocation systems disallow cross-socket sharing of memory among consolidated applications, the memory bandwidth resources of multi-socket hosts cannot be properly exploited. Therefore, this thesis aims at filling that gap by designing, implementing and evaluating two novel techniques for memory bandwidth allocation for multi-socket Cloud nodes. First, we propose BWAP, a novel bandwidth-aware page placement tool for memory-intensive applications on non-uniform memory access (NUMA) systems. BWAP takes the asymmetric bandwidths of every NUMA node into account to determine and enforce an optimized application-specific weighted interleaving. Our evaluations on a diverse set of memory-intensive workloads, show that BWAP achieves up to 4× speedups when compared to a *first-touch* baseline policy (as provided by Linux's default). Second, we propose BALM, a QoS-aware memory bandwidth allocation technique for multi-socket architectures. The key insight of BALM, is to combine commodity bandwidth allocation mechanisms originally designed for single-socket with a novel adaptive cross-socket page migration scheme. Our evaluation shows that BALM can safeguard the SLO of latency-critical applications, with marginal SLO violation windows, while delivering up to 87% throughput gains to bandwidth-intensive best-effort applications compared to state-of-the-art alternatives.

All solutions proposed and presented in this thesis, namely OnlineElastMan, BWAP and BALM, have been implemented and evaluated on real-world workloads. The result indicates the feasibility and effectiveness of our proposed approaches to improve inter-resource and intra-resource management through QoS-aware elastic execution and effective arbitration of resources among consolidated workloads in Cloud nodes.

Keywords

Cloud computing
QoS-aware resource allocation
Elasticity controller
Multi-socket systems
Non-uniform memory access
Page placement
Memory-intensive applications

Sammanfattning

Molntjänster har blivit allmänt förekommande på grund av dess resursflexibilitet och kostnadseffektivitet. Resursflexibilitet gör det möjligt för molnanvändare att skala sina molnesurser elastiskt, till exempel genom att horisontellt skala antalet virtuella maskiner till varje applikation när applikationen kräver förändring. Att matcha resurskrav till applikationer är dock inte trivialt och applikationer som upplever mycket dynamiska arbetsbelastningar gör det mycket svårare. Kostnadseffektivitet uppnås främst genom arbetsbelastningskonsolidering, dvs genom samlokalisering av applikationer på samma fysiska värd. Tyvärr har arbetsbelastningskonsolidering ofta en prestationsstraff, eftersom konsoliderade applikationer strider om delade resurser, vilket leder till störningar och oförutsägbarhet. Störningar är särskilt avledande för latentkritiska applikationer, som måste uppfylla strikta kvalitetskrav (QoS). En annan betydande teknologisk trend är den växande förekomsten av system med flera socklar i samtida datacenter. Såvitt vi vet är emellertid befintliga förslag för QoS-medveten resurstilldelning enligt design inte skräddarsydda för den växande förekomsten av multisockelsystem i samtida datacenter på varuhusskala. Specifikt stöder befintliga förslag inte korssockeldelning av av minne, vilket innebär en suboptimal användning av multisockelvärdens sammanlagda minnesresurser.

Denna avhandling fokuserar på två aspekter av molnresurshantering, nämligen QoS-medveten elasticitet och resursarbitering, på två nivåer: inter-nod resurshantering och intra-nod resurshantering. I den första nivån betraktar vi antalet virtuella maskiner (VM) som den viktigaste resursen för att allokera och avdela för horisontell automatisk skalning av en elastisk tjänst eller applikation i molnet. I resurshanteringen inom noden behandlar vi minnesbandbredden i multisockelsystem som en resurs för att arbitrera bland samplacerade applikationer. På båda nivåerna är det övergripande målet för denna avhandling att tillhandahålla resurshanteringsmekanismer som automatiskt anpassar de resurser som allokeras till dataintensiva tjänster för att förbättra resursutnyttjandet samtidigt som man når servicenivåmål (SLO).

Inom ramen för resurshantering mellan noder för automatisk skalning av elastiska molntjänster förbättrar denna avhandling användbarheten av elasticitetsstyrenheter genom att ta itu med några av de utmaningar som nuvarande modell-förutsägbara styrsystem (såsom utbildning och inställning av styrenheten anpassa den till olika arbetsbelastningsmönster). För att möjliggöra elastisk körning av molnbaserade tjänster med hjälp av modellförutsägbar kontroll, föreslår vi, implementerar, och utvärderar OnlineElastMan, en självutbildad proaktiv elasticitetshanterare för molnbaserade lagringstjänster. OnlineElastMan utmärker sina kamrater med sina praktiska aspekter, inklusive lätt mätbara

och uppnåbara prestanda och QoS-mätvärden, automatisk onlineutbildning och en inbäddad generisk arbetsbelastningsmodul. Vår utvärdering visar att OnlineElastMan kontinuerligt förbättrar sin leveransnoggrannhet, minimerar provisioneringskostnader och SLO-överträdelser, under olika arbetsbelastningsmönster.

I samband med resurshantering inom noder avviker denna avhandling från iakttagelsen att eftersom toppmoderna QoS-medvetna resurstilldelningssystem inte möjliggör delning av minne mellan olika konsoliderade applikationer, är minnesbandbreddens resurser multi-socket-värdar kan inte utnyttjas ordentligt. Därför syftar denna avhandling till att fylla det gapet genom att utforma, implementera och utvärdera två nya tekniker för allokering av minnesbandbredd för molnnoder med flera socklar. Först föreslår vi BWAP, ett nytt bandbreddsmedvetet sidplaceringsverktyg för minnesintensiva applikationer på icke-enhetligt minnesåtkomstsystem (NUMA). BWAP tar hänsyn till de asymmetriska bandbredderna för varje NUMA-nod för att bestämma och genomdriva en optimerad applikationsspecifik viktad sammanflätning. Våra utvärderingar av en mängd olika minnesintensiva arbetsbelastningar visar att BWAP uppnår upp till fyra gånger snabbare jämfört med en grundläggande princip för första tryck (som tillhandahålls av Linux standard). För det andra föreslår vi BALM, en QoS-medveten teknik för tilldelning av minnesbandbredd för arkitekturer med flera socklar. Nyckelinsikten med BALM är att kombinera mekanismer för tilldelning av råvaror för bandbredd som ursprungligen designades för enkelsockel med ett nytt adaptivt korssockel sidmigrationsschema. Vår utvärdering visar att BALM kan skydda SLO för latentkritiska applikationer, med marginella SLO-överträdelsefönster, samtidigt som det ger upp till 87% genomströmningsvinster till bandbreddsintensiva bästa-försöksapplikationer jämfört med toppmoderna alternativ.

Alla lösningar som föreslagits och presenteras i denna avhandling, nämligen OnlineElastMan, BWAP och BALM, har implementerats och utvärderats på verkliga arbetsbelastningar. Vår experimentella utvärdering indikerar genomförbarheten och effektiviteten av våra föreslagna tillvägagångssätt för inter-resurs- och intra-resurshantering som syftar till att förbättra resursutnyttjandet genom QoS-medveten elastisk körning i molnet och effektiv skiljedom av resurser mellan konsoliderade arbetsbelastningar i molnnoder.

Nyckelord

Molntjänster
QoS-medveten resursallokering
Elasticitetsstyrenhet
Flersocklade system
Ojämn minnesåtkomst
Sidplacering
Minneintensiva applikationer

Resumo

A Computação na Nuvem tornou-se onipresente devido à sua flexibilidade de recursos e eficiência de custos. A flexibilidade de recursos permite que os utilizadores da Nuvem escalonem elasticamente os recursos da Nuvem que utilizam – por exemplo, escalando horizontalmente o número de máquinas virtuais alocadas a cada aplicação – à medida que a sua carga de trabalho (e a consequente necessidade de recursos) varia. A eficiência de custos é alcançada principalmente por meio da consolidação da carga de trabalho, ou seja, colocando múltiplas aplicações no mesmo nó físico.

Esta tese foca-se em dois aspectos da gestão de recursos de Nuvem, a saber, elasticidade e arbitragem de recursos, na presença de requisitos de QdS, em dois níveis: a gestão de recursos entre nós e a gestão de recursos dentro de cada nó. No nível inter-nó, consideramos o número de máquinas virtuais como o principal recurso a alocar e desalocar para escalonamento automático horizontal de um serviço ou aplicação elásticos na Nuvem. Mais precisamente, esta tese aborda alguns dos desafios práticos associados aos controladores de elasticidade que recorrem a controlo preditivo baseado em modelação. Mais precisamente, propomos, implementamos e avaliamos o OnlineElastMan, um gerenciador de elasticidade proativo auto-treinado para serviços de armazenamento baseados na Nuvem. Em relação ao Estado da Arte atual, o OnlineElastMan introduz vantagens práticas, nomeadamente por se basear em métricas de carga de trabalho e QdS que são fáceis de obter em sistemas reais, por suportar o auto-aperfeiçoamento automático e dinâmico dos modelos de controle, assim como um módulo genérico de previsão de carga de trabalho. A avaliação experimental do OnlineElastMan mostra que a nossa proposta melhora continuamente a sua precisão de provisionamento, minimizando os custos associados e as violações de QdS, sob vários padrões de carga de trabalho.

No nível intra-nó, esta tese trata a largura de banda da memória em sistemas *multi-socket* como o recurso essencial a arbitrar entre aplicações co-localizadas. A tese parte da observação de que, como os sistemas de alocação de recursos de última geração não permitem que aplicações residentes no mesmo nó partilhem memória entre *sockets*, a largura de banda de memória total de nós *multi-socket* não pode ser completamente explorada. Esta tese visa preencher essa lacuna desenhando, implementando e avaliando duas novas técnicas para alocação de largura de banda de memória para nós *multi-socket* na Nuvem. Primeiro, propomos BWAP, uma nova ferramenta de posicionamento de páginas de memória que optimiza a largura de banda de memória em sistemas de acesso não uniforme à memória (NUMA, em inglês). A avaliação experimental, feita recorrendo a um conjunto diversificado de cargas de trabalho com uso intensivo de memória, mostra que a BWAP

melhora o desempenho até 4× quando comparada com a política trivial *first-touch* (a opção por omissão em Linux). Em segundo lugar, propomos BALM, uma técnica de alocação de largura de banda de memória baseada em QdS para arquiteturas *multi-socket*. A ideia essencial da BALM é combinar mecanismos de alocação de largura de banda existentes em sistemas reais, originalmente projetados para uso em nós de *socket* único, com um novo esquema de migração de páginas entre *sockets*. Experimentalmente, mostramos que, em comparação com o Estado da Arte, a BALM consegue cumprir os requisitos de QdS das aplicações dessa categoria, apenas com períodos marginais de violação temporária; ao mesmo tempo que alcança ganhos de débito até 87% para as restantes aplicações sem requisitos de QdS que correm no mesmo nó *multi-socket*.

Todas as soluções propostas e apresentadas nesta tese, nomeadamente OnlineElastMan, BWAP e BALM, foram implementadas e avaliadas em cargas de trabalho do mundo real. A avaliação experimental descrita neste documento demonstra que as contribuições permitem melhorar a utilização de recursos da Nuvem através da execução elástica e da arbitragem eficaz de recursos entre cargas de trabalho consolidadas num mesmo nó, tudo tendo em conta os requisitos de QdS.

Palavras-chave

Computação em Nuvem
Alocação de recursos baseada em QdS
Controlo de elasticidade
Sistemas *multi-socket*
Acesso não-uniforme à memória
Alocação de páginas de memória
Aplicações com uso intensivo de memória

Acknowledgments

This thesis could not have been completed without the support of many people who crossed my path during the last few years. I am forever grateful to them.

First and foremost, I express my kind thanks to my advisors Prof. João Barreto (IST) and Prof. Vladimir Vlassov (KTH), for their continuous guidance, constant feedback, motivation and unconditional support throughout the course of this PhD. I warmly thank them for the many hours invested in methodologically training me to conduct research, to write clearly, and to present ideas neatly. I am deeply honored for the chance to work, study, and learn under their supervision.

During my Ph.D., I had the opportunity to collaborate and learn from other talented researchers and faculty. I would like to warmly thank my co-authors, Ahmad Al-Shishtawy, Ying Liu, João Neto, Paolo Romano, Rodrigo Rodrigues, Vivien Quema, Pramod Bhatotia, and Reza Karimi, for their effort and insight to improve my work. I would also like to express my gratitude to the faculty of the Distributed Systems Group (GSD) at INESC-ID for their willingness to share their knowledge and for the many times their valuable advice helped me shape and improve my research. I also thank Amin Mohtasham for sharing his initial findings, which inspired us to pursue part of this work.

I would like to thank all my friends and colleagues at GSD and KTH for their companionship and support. I am grateful for all the wonderful times we had together over the last few years, including the many meals, drinks, and coffee breaks. In no particular order, I thank: Paolo Laffranchini, Igor Zavalyshyn, Sileshi Yalew, João Loff, Paulo Silva, Ricardo Filipe, Pedro Raminhas, Manuel Bravo, Sana Imtiaz, Zainab Abbas, Tianze Wang, Sina Sheikholeslami, Daniel Castro, Diogo Barradas and Daniel Porto. A special thanks goes to Klas Segeljakt for helping me with the Swedish abstract.

Thanks to Stephen Kimogol for proofreading this thesis and pointing out any grammatical and spelling issues.

I am deeply thankful to my family for their love and support throughout my life. Thanks to my parents for their constant love, prayers, and support. I could not have gone this far without them. Special thanks to my love Sarah, who has lived this journey together with me. This work could not have been complete without her support and patience. Above all, I thank her for helping me raise our lovely daughter Ellie.

Finally, I would like to thank the Erasmus Mundus Joint Doctorate in Distributed Computing program (funded by the EACEA of the European Commission under FPA 2012-0030), and Fundação para a Ciência e a Tecnologia (FCT) (via the projects UIDB/50021/2020, and PTDC/EEI-SCR/1743/2014) for funding most of my doctoral research.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The business models for Cloud and data center computing emphasize *resource flexibility* and *cost efficiency* [2–4]. Resource flexibility allows Cloud users to elastically scale the compute, storage and network resources, as the need arises. Since users only pay for the resources used to serve their demand, elastic provisioning enables users reduce the cost of hosting their applications in the Cloud, while meeting requirements on quality of service (QoS), performance, availability and scalability of the applications. The changing demands of each application can be handled by efficiently managing the available resources. Assuming a multi-node distributed infrastructure of the Cloud or a data center, resource management can be achieved in two levels: inter-node (*across* nodes) and intra-node (*within* individual nodes) resource management. Inter-node resource management involves provisioning additional nodes to run more virtual machines (VMs) to meet applications demands, often distributing workloads among VMs to achieve high performance, scalability, and availability of services. Often, this is referred to as horizontal scaling (scaling out or in). In contrast, intra-node resource management is performed in the context of each individual Cloud node (in a bare-metal server or in VM).

Self-management, such as self-healing or auto-scaling, in a Cloud-based service can be achieved by using autonomic managers that monitor the workload and service performance and act whenever needed to meet their management objectives [5]. Typically, each elastic service or application deployed in the Cloud, such as an elastic key-value storage service, has its own elasticity manager. The elasticity manager monitors workload or/and some

service-level objective (SLO) metrics, e.g., service latency, and auto-scales (expands or shrinks) the service by requesting or releasing resources (e.g., VMs or virtual hardware resources assigned to each VM) used by the service, in response to changes in its workload or SLO metrics. In this way, the service only uses the amount of resources that it needs to handle its workload and to meet its SLOs.

In addition, cost efficiency in the Cloud is also achieved through *workload consolidation*, i.e., by co-locating applications on the same physical host. Workload consolidation also improves resource utilization. In a Cloud environment, scheduling jobs from multiple users on the same physical host increases utilization and reduces costs (rental costs and power costs). Contemporary multicore nodes used in data centres and Cloud are designed to allow clusters of cores to share various hardware resources, such as last-level caches (LLCs), memory controllers, and interconnects. These per-node resources, including cores, need to be managed efficiently to ensure consistent performance and QoS, especially in a multitenant environment [6]. Since different co-located applications may now contend for limited shared resources, managing intra-node resources becomes an arbitration challenge (more than an allocation one).

Among the co-located applications, some have QoS requirements, as determined by one or more service-level objectives (SLOs). As typically SLOs include latency requirements and constraints, such applications are commonly called *Latency-Critical* applications (LCAs). In contrast, an application with no QoS requirements determined by SLOs, is meant to run in some best-effort fashion that aims to maximize its throughput (minimize its execution time). In the context of resource allocation, such applications with no SLOs are usually called *Best-Effort* applications (BEAs).

The co-located applications contend for shared resources, such as network and storage bandwidth, CPU cores, LLC, and memory. Therefore, *noisy neighbour* phenomena are prone to arise, in which the demand that some applications place on some shared resources degrade the performance of other co-located applications, up to a point where LCAs start suffering from SLO violations. 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 the LCAs while maximizing the throughput of the BEAs.

Apart from aforementioned workload consolidation, another significant technological trend is the growing prevalence of non-uniform memory access

(NUMA) systems in contemporary data centers. In many warehouse-scale data centers, dual-socket (or even larger) machines already constitute the largest share of hosts [7]. Contention for shared resources on multicore processors is a well-known problem. Studies have shown that for a large class of memory-intensive applications, *memory bandwidth* has been shown to be the most performance-critical shared resource, especially in NUMA systems [8–12]. To prevent performance interference among the consolidated workloads, the importance of efficient partitioning techniques for memory bandwidth is expeditiously increasing [13, 14]. To the best of our knowledge, proposals for memory bandwidth partitioning in scenarios of workload consolidation only focus on single-socket systems [10, 14–16]. Consequently, this research focuses on memory bandwidth allocation in NUMA systems. The studies on other resources (e.g., cache, memory and cores) can be used in conjunction with this research to provide a comprehensive solution for the resource allocation problem in NUMA systems. The studies on extending our memory bandwidth management framework to other resources is a subject to future work.

This thesis focuses on two fundamental aspects of Cloud resource management, namely QoS-aware elasticity and resource arbitration, in the context of the *infrastructure as a service (IaaS)* Cloud service model, where virtualized computing resources are provided in the form of virtual machines (VMs). The thesis focuses on two levels: inter-node resource management and intra-node resource management. In the first level, we consider the number of virtual machines (VMs) as the resource to allocate and de-allocate for horizontal auto-scaling of an elastic service or application in the Cloud. Specifically, VMs are added when needed to handle an increasing workload and removed when the workload drops. Hence, the QoS of an application can be satisfied by scaling out or in (adding or removing VMs) in response to changes in its workload. In resource management on the second level, i.e., in the intra-node resource management, we treat the memory bandwidth in a NUMA system as the resource to arbitrate (allocate and de-allocate) among co-located applications. This NUMA system can be a node within a Cloud. In this level, the memory bandwidth of a NUMA host is partitioned among the co-located applications in order to maintain a desired level of QoS for LCAs while maximizing the throughput of BEAs. In both levels, the overall goal of this thesis is to provide resource management mechanisms that automatically adapt the resources allocated to data-intensive services to improve resource utilization while meeting SLOs.

In the context of inter-node resource management for auto-scaling of elastic Cloud services, this thesis improves the usefulness of elasticity controllers by addressing some of the challenges posed by current model-predictive control systems (such as training and tuning of the controller and adapting it to different workload patterns). In particular, the thesis examines the usefulness of an elasticity controller while deploying it in a Cloud environment.

In the context of intra-node management of memory bandwidth in a NUMA Cloud node, this thesis advocates that, 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 to allow cross-socket sharing of memory among consolidated applications. To achieve that, the memory partitioning mechanisms on which existing solutions rely need to be redesigned to address the new constraints of multi-socket architectures.

1.1 Thesis statement

In inter-node Cloud resource management, well-designed elasticity managers using model-predictive control and online model training enable elastic execution in the Cloud that improves resource utilization while meeting SLOs of the Cloud-based services and applications. In addition, elastic auto-scaling of Cloud-based services with elasticity managers optimizes the overall cost of Cloud resources used by these services.
In intra-node Cloud resource management, well-designed QoS-aware memory bandwidth partitioning techniques can ensure marginal SLO violation windows while delivering better performance for bandwidth-intensive best-effort applications on multi-socket Cloud nodes.

1.2 Problem Definition and Research Questions

In this section, we briefly present the key problems that need to be addressed when allocating VMs in the Cloud for horizontal scaling and when allocating memory bandwidth in a NUMA Cloud node.

VM allocation in the Cloud for horizontal scaling. An elastic application can be scaled to adapt to changing workloads. Applications of this nature typically include a load balancer (dispatcher or master) and a collection of identical servers (workers). In a cluster environment, those would

be physical servers but, in Cloud environments, servers are hosted in VMs. Therefore, the terms servers and VMs can be used interchangeably.

The VMs reside on physical machines (PMs) in a Cloud. We aim to minimize the service provisioning cost while maintaining the desired SLOs through horizontal scaling, i.e., dynamically acquire and release VMs to accommodate varying application loads. Often, horizontal scaling is conducted automatically by an elasticity controller.

Recent works have focused on improving elasticity controllers' accuracy by building different control models with various monitored/controlled metrics [17–24]. However, to the best of our knowledge, none of such works have considered the practical usefulness of a model-predictive elasticity controller, which involves the following challenges. First, an elasticity controller usually needs to be tailored according to a specific application. Sometimes, it requires complicated instrumentations to the provisioned application or it is difficult or even impossible to obtain the metrics to build the control model. Furthermore, even with all the metrics, it needs a tremendous and tedious effort to define and train the control model [25]. A general training procedure involves redeploying and reconfiguring the application and collecting and analyzing data by running various workloads against many application configurations. Second, the hosting environment of the provisioned application may change due to some unmonitored factors, for example, platform interference or background maintenance tasks. Hence, even with well-trained control models, the elasticity controller may not adjust well to these factors and therefore lead to inaccurate control decisions [24, 26]. Third, it is always too late for the elasticity controller to react to a workload increase when the workload is already saturating the application [23]. Thus, we argue that the prediction of the workload is a compulsory element to an elasticity controller [27].

Memory bandwidth allocation in a NUMA node (Cloud node). Although the *QoS-aware resource allocation problem* is not new, the recent emergence of novel hardware-based resource partitioning mechanisms has unveiled the opportunity for a new generation of QoS-aware resource allocation approaches. One notable example of such mechanisms is the support for hardware-based partitioning of LLC and memory bandwidth as provided by Intel Resource Director Technology (RDT) [28]. Recent proposals such as PARTIES [15] and CLITE [16] take advantage of such new mechanisms

to enforce QoS-aware resource allocation with unprecedented effectiveness. Unfortunately, PARTIES and CLITE, as well as the vast majority of their predecessors, are, by design, tailored to single-socket architectures only.

A common approach to deploy such solutions in a multi-socket host is to distribute applications among sockets [9, 29, 30] and consider the resources contained in each socket (CPU cores, LLC, memory) as exclusively shared among the applications in that socket. This way, each socket can be considered as an independent workload consolidation island, which can be managed by an independent instance of some QoS-aware resource allocation system. The only resources that require cross-socket allocation are external, OS-managed resources such as disk or network [15].

While simple in practice, this strict approach prevents an application running in a given socket to place data pages on remote memory nodes (from other sockets). This essentially disallows cross-socket sharing of memory, which entails a sub-optimal use of multi-socket host's aggregate memory resources. This limitation is especially relevant given the increased prevalence of memory-intensive applications among data center workloads, whose performance strongly depends on how efficiently memory bandwidth is allocated [9, 14, 29].

As an example, consider the case where a memory-intensive best-effort application (BEA) runs in one socket and saturates the local memory bandwidth, while a CPU-intensive latency-critical application (LCA) runs on another socket and only places a negligible access demand on the local memory. Clearly, allowing the bandwidth-intensive BEA to place a portion of its pages in the idle remote memory node in order to benefit from an improved aggregate memory bandwidth, while not causing any harmful interference with the LCA, would be considerably more efficient.

Furthermore, this problem is inherently dynamic, 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 within negligible SLO violation windows.

Research Questions. In summary, this thesis aims at addressing two main research questions.

1. *How can the number of VMs allocated to applications be adjusted efficiently to the currently experienced workload and anticipated future*

workload?

We address this research question by focusing on the specific class of distributed storage systems, one of the most important types of services that are deployed in the Cloud. More precisely, the following two main design issues need to be considered in order to address and answer the above research question.

a) *How to build an "out-of-the-box" generic elasticity controller framework?*

b) *How to design a generic workload prediction module, which is adjustable to multiple workloads?*

To address the first issue, we aim at building an elasticity controller that is applicable to most distributed storage systems that scale horizontally. This controller should be deployed and adopted by different applications without complicated tailoring/configuring efforts. Furthermore, this research problem is of practical nature and aims at evaluating whether the proposed elasticity manager can be implemented efficiently with real Cloud-based applications. To address the second design issue, we aim to make the elasticity controller proactive, such that it can provision VMs in advance and avoid performance degradation. Stateful services such as storage systems need to be initialized with data. Spawning/removing VMs also takes a considerable amount of time. Therefore, it is always too late for the elasticity controller to react to a workload increase when the workload is already saturating the application. Previous works [18, 23] have demonstrated that, in order to keep the SLO commitment, a storage system needs to scale out in advance to tackle a workload increase since scaling a storage system involves non-negligible overhead. To make the workload prediction module as general as possible, it should produce accurate workload prediction for various workload patterns. Hence, the performance of a proactive controller largely depends on the accuracy of workload prediction, which varies for different workload patterns.

2. *How to elastically adjust the memory bandwidth of a multi-socket NUMA host allocated to data-intensive services to improve resource utilization while meeting SLOs of QoS-based services and maximizing throughput of best-effort applications?*

In order to address the above research question, the following two main research issues need to be considered.

a) *How to allocate memory resources of a NUMA host in a way that optimizes the throughput of a given application?*

b) *How to design efficient QoS-aware memory bandwidth allocation for multi-socket Cloud nodes?*

Regarding the first issue, contemporary NUMA systems are characterized by asymmetric bandwidths and latencies. When one deploys a parallel application on a NUMA system, its threads allocate and access pages that need to be physically mapped to the available NUMA nodes. This raises a crucial question: *where should each page be mapped for optimal performance?* When the application is memory-intensive, a common strategy is to uniformly interleave pages across the set of *worker* nodes, i.e., the nodes on which the application threads run. This strategy is based on the rationale that, for a large class of memory-intensive applications, bandwidth – rather than access latency – is the main bottleneck. Therefore, interleaving pages across nodes provides threads with a higher aggregate memory bandwidth [11]. However, this approach fails to maximize *memory throughput* in modern asymmetric NUMA systems by considerable margins. Furthermore, as noted by different authors [31, 32], the memory bandwidth of a NUMA system is particularly hard to determine accurately, since it is sensitive to interconnect congestion and local-remote contention on memory controllers phenomena which, in turn, depend on the memory demand patterns of the deployed application(s). Hence, optimal placements are eminently application-specific.

With respect to the second issue, as mentioned earlier, to properly utilize over-provisioned memory resources in multi-socket hosts, state-of-the-art QoS-aware resource allocation systems need to be generalized to allow cross-socket sharing of memory. This generalization requires additional extensions to embrace the additional complexity of multi-socket workload consolidation scenarios. Therefore, several challenges have to be addressed to support such scenarios. Striking a balance between optimizing BEAs throughput and satisfying LCAs SLO most of the time is also challenging, especially under highly dynamic workloads. Runtime solutions inevitably add overhead, as they present an

extra layer into the system to monitor and manage executing applications. For example, application memory behavior profiling and migrating large amounts of memory can be extremely costly. Therefore, it is important that these operations are done in a way that minimizes overhead as much as possible.

1.3 Research Methodology

The work of this thesis is based on empirical studies conducted by designing and implementing proof-of-concept prototypes of proposed solutions, evaluated by experiments on real-world workload. Specifically, we first quantify an identified problem and propose a solution. Then, we evaluate the proposed solution and analyze the results. This section describes the approach we followed in this dissertation to conduct the research work, including problem definition, design and evaluation of solutions, and the challenges faced throughout the thesis work and how we have addressed them.

We approach the problem by first studying related works and literature. We have studied the state-of-the-art approaches for VM allocation in the Cloud for horizontal scaling and memory bandwidth allocation in NUMA. Consequently, we have identified important limitations of the approaches and algorithms that define the state-of-the-art in the fields of our research, namely, resource allocation and management in the Cloud and NUMA Cloud nodes. The studied approaches and algorithms enable elastic execution of Cloud services while meeting SLOs, as well as improve performance and resource utilization. We have exposed open issues to be addressed and unexplored directions towards better approaches. We justify and empirically verify the existence of the problems by running extensive experiments in real-world settings. For example, in BWAP (Chapter 4), we start by empirically studying the performance for different page placement strategies on a range of memory-intensive applications from different domains on different NUMA machines.

We implement our proposed solutions from scratch or augment existing solutions whenever possible. All our implementations are open source and are publicly available. The links to the public repositories are available in the corresponding papers and also provided in the chapters of the thesis. Experiments were performed on real systems and platforms, including multi-socket NUMA servers and a private Cloud that runs the OpenStack software stack.

The experiments were executed using scripts to ensure the reproducibility of experiments. The experiments were conducted using benchmarks and real-life applications. The benchmarks consisted of memory-intensive applications from several benchmark suites, i.e., NAS [1], PARSEC [33] and SPLASH [34]. These benchmarks represent a wide diversity of application domains, which are typically throughput-oriented, and they are also used as such in related work on QoS-aware resource allocation. Real-life applications such as Memcached [35], Cassandra [36] and Xapian [37] were used as the LCA workloads in our experiments. To compare performances, LCAs SLO violation times and BEAs execution times were measured and collected. Other than that, backend-bound stall cycles, memory access patterns, and memory bandwidth consumption were collected to provide a more in-depth understanding of the results presented in Chapter 4 and 5. The results presented in this thesis are the average of at least 5 runs with a standard deviation of less than 2% of the mean value.

1.4 Thesis Contributions

The two main research questions discussed in Section 1.2 are addressed via three novel contributions, which are presented in this section.

OnlineElastMan. We address the first research question of enabling and achieving elastic execution of Cloud-based services using QoS-aware model-predictive control by proposing, implementing and evaluating OnlineElastMan, that is a self-trained proactive elasticity manager for Cloud-based storage services. OnlineElastMan excels its peers with its practical aspects, including easily measurable and obtainable performance and QoS metrics, an automatically online trained control model, and an embedded generic workload prediction module. This makes OnlineElastMan an "out-of-the-box" elasticity controller, which can be deployed and adopted by different storage systems without complicated tailoring/configuring efforts. Specifically, OnlineElastMan requires monitoring the two most generic metrics, i.e., incoming workload and service latency, which are obtainable from most storage systems without complicated instrumentation. Using the monitored metrics, OnlineElastMan analyzes the workload composition in-depth, including read/write request intensity and the requested item's data size, which define the dimensions of a control model. OnlineElast-

MAN can easily plug in more interesting dimensions if needed. After fixing the dimensions, a multi-dimensional control model can be automatically built and trained online while the storage system is serving requests. After a sufficient amount of warm-up on the control model, ONLINEELASTMAN can issue accurate control decisions based on the incoming workload. Furthermore, the control model continuously improves itself online to adjust to unknown/unmodeled events of the operating environment. Additionally, a generic workload prediction module is also integrated to facilitate the decision making of ONLINEELASTMAN. It allows ONLINEELASTMAN to scale the storage system well in advance to prevent SLO violations caused by workload increase and scaling overhead [23]. Specifically, the prediction module aggregates multiple prediction algorithms and chooses the most appropriate prediction algorithm based on the current workload pattern using a weighted majority selection algorithm. Experiments using ONLINEELASTMAN with Cassandra indicate that ONLINEELASTMAN continuously improves its provision accuracy, i.e., minimizing provisioning cost and SLO violations, under various workload patterns. This contribution is presented in Chapter 3.

BWAP is a novel bandwidth-aware page placement tool for memory-intensive applications on NUMA systems. Page placement is one way of allocating memory bandwidth resources in NUMA systems. Therefore, we start by empirically studying the performance for different page placement strategies on a range of memory-intensive applications from different domains on different NUMA machines. Our findings i) shed new light on the page placement problem, showing that common practices that rely on the obsolete assumption of a symmetric architecture are largely sub-optimal on contemporary NUMA systems, ii) expose unexplored directions towards better page placement strategies. To overcome these inefficiencies, we propose BWAP. In contrast to existing solutions, BWAP takes the asymmetric bandwidths of every NUMA node into account to determine and enforce an optimized application-specific *weighted interleaving*. We experimentally evaluate BWAP on a diverse set of memory-intensive workloads, from PARSEC[33], SPLASH [34] and NAS [1], showing that BWAP achieves up to 4× speedups when compared to a *first-touch* baseline policy (as provided by Linux's default). This represents a 66% improvement over the performance gains that the most commonly used placement policies attain over the same baseline. These benefits are particularly relevant in scenarios where multiple

co-scheduled applications run in disjoint partitions of a large NUMA machine or when applications do not scale up to the total number of available cores. To the best of our knowledge, this is the first proposal for bandwidth-aware page placement in heterogeneous memory systems to be evaluated on real commodity machines, i.e., not based on simulation [38–40]. BWAP is presented in Chapter 5.

BALM is a QoS-aware memory bandwidth allocation technique for multi-socket architectures. As a first contribution, we study the hardware mechanism of Intel RDT for memory bandwidth allocation (MBA) in a multi-socket scenario (MBA being the state-of-the-art mechanism for memory bandwidth allocation). We show that, in order to use MBA to fix harmful memory bandwidth interferences between a bandwidth-intensive BEA and an LCA, one may need to limit the memory bandwidth of the BEA (via MBA) well beyond the limit that an optimal allocation would require. As we detail in Chapter 5, this limitation can mean an unnecessary throughput reduction of BEAs by a considerable margin, relatively to an optimal allocation. This limitation constitutes a fundamental limitation that restricts the ability of state-of-the-art QoS-aware resource allocation systems to embrace the power of multi-socket hosts fully. BALM is, to the best of our knowledge, the first that address this open question. The key insight of BALM is to combine commodity bandwidth allocation mechanisms originally designed for single-socket with a novel adaptive cross-socket page migration scheme. By doing so, BALM can overcome 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. Our experimental evaluation with real applications on a dual-socket machine shows that BALM can overcome the efficiency limitations of state-of-the-art. BALM can safeguard the SLO of LCAs, with marginal SLO violation windows, while delivering up to 87% throughput gains to bandwidth-intensive BEAs compared to state-of-the-art alternatives. BALM is detailed in Chapter 5.

1.5 List of Publications

Most of the contents of this thesis is based on the following papers, which have been peer-reviewed.

I **David Gureya** and João Barreto. Profiling for Asymmetric NUMA Systems. 11*th* EuroSys Doctoral Workshop (EuroDW 2017).

II **David Gureya**, Rodrigo Rodrigues, Paolo Romano, Pramod Bhatotia, Vivien Quéma and João Barreto (2018, April). Asymmetry-aware Page Placement for Contemporary NUMA Architectures. 8*th* Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA 2018).

III Ying Liu, **Daharewa Gureya**, Ahmad Al-Shishtawy and Vladimir Vlassov. OnlineElastMan: Self-Trained Proactive Elasticity Manager for Cloud-Based Storage Services. In 2016 International Conference on Cloud and Autonomic Computing (ICCAC), pages 50–59, 2016.

IV Ying Liu, **Daharewa Gureya**, Ahmad Al-Shishtawy, and Vladimir Vlassov. Onlineelastman: Self-trained proactive elasticity manager for cloud-based storage services. Cluster Computing, 20(3):1977–1994, September 2017.

V **D. Gureya**, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov. Bandwidth-Aware Page Placement in NUMA. In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 546–556, 2020.

VI **David Gureya**, João Barreto and Vladimir Vlassov. Brief Announcement: BALM: QoS-Aware Memory Bandwidth Partitioning for Multi-Socket Cloud Nodes. In 2021 ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'21), 2021

VII **David Gureya**, João Barreto and Vladimir Vlassov. Generalizing QoS-Aware Memory Bandwidth Allocation to Multi-Socket Cloud Servers. To appear in IEEE International Conference on Cloud Computing (IEEE CLOUD'21).

The content of Chapter 3 is based on Papers II and III, while the content of Chapter 4 is based on Papers I, II and V, and the content of Chapter 5 is based on Paper VI and Paper VII.

For all the papers, the author of the thesis is the main contributor of the work. First, he contributed to the conception and development of the idea. Second, the author of the thesis led and was responsible for the prototype implementations and carrying out the experimental evaluation using synthetic micro-benchmarks and real applications. Lastly, he was a major contributor to the writing of the paper.

1.6 Dissertation Outline

The rest of this thesis is organized as follows: Chapter 2 provides the background information and the related works to this thesis. The subsequent chapters present the detailed contributions of this thesis. Specifically, Chapter 3 explains OnlineElastMan, then BWAP is discussed in Chapter 4 and Chapter 5 presents BALM. Finally, Chapter 6 summarizes our work and presents possible directions for future work.

Chapter 2

Background and Related Work

This chapter presents the background and the related work of this dissertation. As mentioned in Chapter 1, this thesis presents solutions for resource allocation in the Cloud with a focus on inter-node and intra-node resource management. Therefore, this chapter presents the concepts, theories, and tools used in this dissertation in both scopes, inter-node and intra-node resource management. Regarding inter-node resource management, our focus is on VM allocation for autoscaling, while in intra-node resource management, we focus on memory bandwidth allocation in a NUMA machine.

2.1 Cloud Computing and Elastic Services

According to the National Institute of Standards and Technology (NIST), Cloud computing is defined as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [41].

This model highlights five essential features [42], which we briefly present below:

- *On-demand self-service*: The ability to automatically provide computational resources such as service time and network storage when

needed.

- *Broad network access:* Capabilities are delivered over the network and accessed via standard mechanisms, allowing heterogeneous thin or thick client platforms to make use of the computational resources.
- *Resource pooling:* Cloud subscribers are served by pooling the Cloud provider's resources in a multi-tenant model where different physical and virtual resources are dynamically assigned or reassigned to subscribers based on their demand. The resources include storage, processing, memory, network bandwidth among others. Additionally, details such as resource location, specific configurations, and failures are abstracted from the subscriber.
- *Rapid elasticity:* Cloud services can be elastically provisioned and released, in some cases automatically, to quickly scale in and out based on demand. The Cloud provider provide an illusion of unlimited resources, allowing customers to request resources in any quantity at any time.
- *Measured service:* Cloud resource usage could be monitored, controlled, and reported to provide transparency to both the Cloud provider and the user of the service. In cloud computing, a metering capability [1] is used to control and optimize resource use. Just like utility companies sell services such as municipality water or electricity to subscribers, Cloud services are also charged per usage metrics - pay as you go. Cloud operators utilize a consumption-based model i.e., the more a resource is utilized, the higher the bill. To keep consumers satisfied with a system, it is important to keep real time constraints on its performance without compromising service quality.

Furthermore, NIST's definition of Cloud computing considers three main service models:

- *Infrastructure-as–a-Service (IaaS):* Provides information technology and network resources such as processing, storage, and bandwidth, as well as management middleware. Examples include Amazon EC2 [3] and Google Compute Engine [4]. In this thesis, we focus on the *IaaS* model.

[1] Typically this is done on a pay-per-use or charge-per-use basis

- *Platform-as-a-Service (PaaS):* Provides programming environments and tools hosted and supported by Cloud providers that can be used by consumers to build and deploy applications onto the Cloud infrastructure.
- *Software-as-a-Service (SaaS):* Provides hosted vendor applications.

Cloud computing, with its pay-as-you-go pricing model, provides an attractive solution to host the ever-growing number of web applications [43]. This is mainly because it is difficult, especially for startups, to predict the future load that might be imposed on the application and thus to predict the amount of resources needed to serve that load. Another reason is that the capital expenditure of buying, powering and maintaining the physical infrastructure is avoided in the Cloud pricing model.

To leverage the Cloud pricing model and to efficiently handle the dynamic workload, Cloud services are designed to be elastic. An elastic service can scale horizontally at runtime by provisioning additional resources without disrupting the service. The benefit of elastic resource allocation to Cloud systems is to minimize resource provisioning costs while meeting service-level objectives (SLOs). With the emergence of elastic services, and more particularly elastic key-value stores, that can scale horizontally by adding/removing VMs, organizations perceive potential in being able to reduce cost and complexity of large-scale Web 2.0 applications.

In the following subsections, we introduce some fundamental notions of Cloud computing.

Virtualization

Cloud service providers usually rely on virtualization-based approaches to build their stack. Virtualization enables multiple operating systems and applications to be run on the same physical server simultaneously, i.e., it creates an abstraction layer that masks the complexity of the hardware. Thus, applications can be deployed and scaled promptly through rapid provisioning of the virtualized resources. This deployment and scaling are done through virtualization techniques, typically virtual machines (VMs) or containers. In the VM alternative, a hypervisor allows multiple operating systems to share a single physical host, such that each operating system appears to have its independent resources. Examples of widely-used hypervisors include VMware ESXi, Xen, Hyper-V, and KVM [44]. In contrast in the containerization

alternative (also called operating system virtualization), the virtualization layer runs as an application within the operating system. One of its main advantages over hardware-based virtualization is that it allows faster start-up time and less overhead [14, 45]. Therefore, containerization is considered as a lightweight virtualization solution. However, the use of containers poses security implications since it is challenging to implement the same level of isolation between containers as VMs do [44].

Cloud service providers have recently introduced the bare-metal Cloud service [46], where a Cloud user can rent dedicated physical servers. Hence, the user has exclusive, full access to the hardware. This provides the user with the same level of performance isolation and security as the PM. However, bare-metal Cloud services have a severe limitation in cost-efficiency, as a PM can only be rented to one user at a time.

Since virtualization is a fundamental part of Cloud computing, in this thesis we discuss Cloud elasticity in the context of VMs.

Service Level Agreement

The QoS that is expected from a Cloud service provider is defined as a Service Level Agreement (SLA). Cloud service providers and Cloud service consumers usually negotiate and agree upon SLAs. SLA can specify the availability aspect and/or performance aspect of a service, such as service uptime and service tail latency. SLA violation affects both the service provider and the consumer. If the service provider violates the SLA, penalties are paid to the consumers. From the consumer's perspective, a violation of SLA can result in degraded service and consequently lead to a decline in profits. Therefore, commitment to SLAs is essential to both Cloud service providers and consumers. An SLA is usually divided into multiple Service Level Objectives (SLOs). SLOs are ways of measuring the performance of a service provider regarding a particular service. SLOs are often quantitative and have related measurements, e.g., service uptime or service tail latency (e.g., 99% of the requests are fulfilled within 100ms).

Stateful and Stateless Services

Cloud services can be divided into two categories: stateless and stateful. Examples of stateless services include front-end proxies and static web servers.

Distributed storage services are typical examples of stateful services, where state/data need to be properly maintained.

Stateless services are intrinsically easy to scale [47], since autoscaling can be done by merely spawning new VMs or deleting existing ones. In contrast, scaling stateful services is challenging since only a particular subset of servers host data of the popular item. For stateful services, scaling usually requires state transfer and/or replication, which adds overhead to the scaling process [19, 23]. Specifically, when scaling up a stateful service (spawning VMs), a VM cannot function until appropriate states are transferred to it. When scaling down a stateful service (removing VMs), a VM's state must be transferred to be handled by other VMs before it can be safely removed. Moreover, if the scaling operations are not effectively handled, this scaling overhead creates an additional workload for the other VMs in the system, which can lead to performance degradation of the system. Thus, scaling a stateful service is challenging.

Many large-scale web applications, such as social networks, wikis, and blogs, are data-centric, requiring frequent data access [48]. They typically leverage stateful elastic services, such as elastic key-value stores, as the data-tier [21, 48]. This poses hard challenges on the data-tier of multi-tier applications because the performance of the data-tier is mostly governed by strict SLOs [49]. With the rapid increase of the number of users, the poor scalability of a typical data-tier with ACID [50] properties limited the scalability of web applications. This has led to the development of NoSQL databases with relaxed consistency guarantees and simpler operations in order to achieve horizontal scalability and high availability. Examples of NoSQL data-stores include, among others, key-value stores such as Voldemort [51], Dynamo [52], and Cassandra [53].

In this thesis, we investigate the elastic scaling of stateful distributed storage systems. Specifically, we focus on key-value stores, which typically provide simple key-value pair storage with weak consistency guarantees. The simplified data and consistency models of key-value stores enable them to efficiently scale horizontally by adding more VMs and thus serve more clients. In a distributed storage system, service latency is one of the most commonly defined SLOs [54]. Satisfying latency SLOs in back-end distributed storage systems that serve interactive LCAs is desirable.

Resource Management

Resource management is an integral part of the software infrastructure of today's data centres [2]. The mapping of user tasks to hardware resources is managed by a resource manager. The resource manager also sets priorities and quotas, as well as providing basic task management. An advanced resource manager would provide a higher level of abstraction, automate resource allocation, and allow sharing of resources at a finer level of granularity. Resource management can be enforced both within *individual* nodes (intra-node resource management) and *across* nodes (inter-node resource management).

Given the high cost of server machines and their rapid depreciation, data centre operators strive to get the most out of their investment. However, production systems are generally underutilized, with just around 30% of their capability being used on average [2]. This is particularly true for data centres that host interactive latency-sensitive services [55, 56]. The reasoning behind the idea is that leaving some headroom aids in achieving low-latency. As a result, in order to achieve their (often) stringent SLOs, such systems purposely over-provision resources to services.

Matching resource demands to application workloads is non-trivial, and highly dynamic workloads make it much more difficult. As a consequence, nodes may end up having resources that will never be used. Furthermore, a contemporary data centre runs a wide range of applications. Therefore, the request demands are diverse, and many nodes may end up having unused resources [2].

Workload consolidation is a practical and widely used technique to improve the resource efficiency of data centres and Cloud computing systems. With workload consolidation, multiple workloads are consolidated on the same physical machine. The primary design goal of the resource manager for workload consolidation is to maximize the server resource utilization while meeting SLOs and providing higher throughput (or trades off throughput vs. fairness [14, 57]) for consolidated applications.

2.2 Inter-node resource management

Cloud data centres are made up of a large number of physical machines (PMs). Cloud service providers offer VMs through virtualization technology

such that customers can run their applications while facilitating the sharing of PMs resources such as CPU, memory, and storage.

In a Cloud data centre, inter-node resource management comprises two essential problems: scaling applications to multiple VMs within the data centre (VM autoscaling) or VM-to-PM allocation/placement. A relevant body of research has addressed inter-node resource management by either focusing on the VM-to-PM packing/placement or resource autoscaling [42, 44, 58]. The goal of VM-to-PM placement is to assign the allocated VMs to PMs in a way that minimizes the operational costs, saves power consumption, and/or generates higher revenues by accommodating more VMs in a single PM. In contrast, the goal of VM autoscaling is to dynamically adjust the number of VMs to meet the QoS requirements when an application's demand for resources changes.

In this thesis, we only consider the VM autoscaling problem.

VM allocation in the Cloud for autoscaling

In the following sections, we lay out the necessary background on VM autoscaling, including elastic scaling, elasticity controllers, and workload prediction.

Elasticity

According to Herbst et al. *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"* [59]. The emergence of large-scale interactive web applications imposes increased challenges to the underlying provisioning infrastructure, such as scalability, highly dynamic load, and partial failures. In order to respond to changes in workload pattern, an elastic service is needed to meet SLOs at a reduced cost. Specifically, VMs are spawned to handle an increasing workload and removed when the workload drops.

Most of the existing elasticity solutions are tailored to the IaaS model, and they are suited for client-server applications [3, 20, 23, 24, 27, 60–63]. In general, IaaS Clouds have an elasticity controller, which is responsible for converting the Cloud user requirements to actions provided by IaaS Clouds. The architecture of an elasticity controller generally follows the

idea of MAPE-K (Monitor, Analysis, Plan, Execute-Knowledge) control loop [64]. The controller uses monitoring data from applications and makes scaling decisions.

In summary, Al-Dhuraibi et al. [44] formulated Equation 2.1, which summarizes the concept of elasticity in Cloud computing. The equation denotes that scalability is associated with elasticity. Scalability refers to the ability of a system to sustain increasing workloads by making use of additional resources [59]. Equation 2.1 indicates that elasticity can be considered as an automation of the scalability concept. Additionally, elasticity aims to optimize the system resources at any given time.

$$Elasticity = \underbrace{scalability + automation}_{\text{auto-scaling}} + optimization \tag{2.1}$$

Elastic Scaling

The goal of an autoscaling system is to automatically fine-tune acquired resources of a system to minimize resource provisioning costs while meeting SLOs. An autoscaling technique automatically scales resources according to demand. Figure 2.1 shows the traditional and elastic provisioning of services under a diurnal workload pattern. In the traditional provisioning approach, a service is provisioned with the amount of resources needed for its maximum load in order to meet SLO at all times. As shown in Figure 2.1 (a), it is clear that this approach wastes a considerable amount of resources when the workload drops from its maximum level, hence resulting in high provisioning cost. Figure 2.1 (b) illustrates elastic provisioning where the amount of provisioned resources follows the changes in the workload. Elastic provisioning saves a significant amount of resources compared to the traditional provisioning approach. However, both approaches aim to guarantee a specific level of SLO. For instance, Figure 2.1 (c) depicts a latency-based SLO compliance that is guaranteed most of the time. The main goal of a well designed provisioning approach is to prevent SLO violations with the minimum amount of provisioned resources, achieving the minimum provisioning cost.

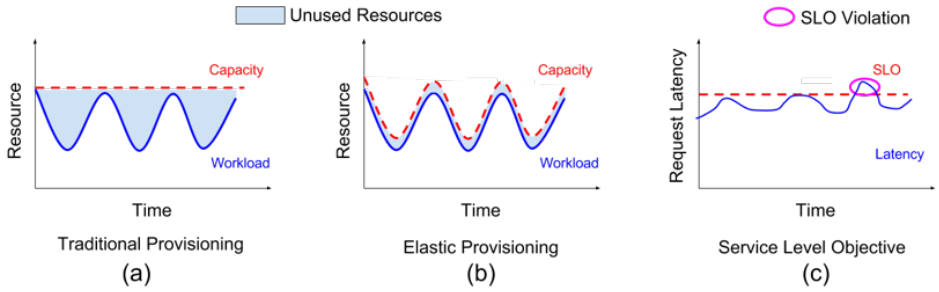


Figure 2.1: (a) - traditional provisioning of services; (b) elastic provisioning of services; (c) latency-based SLO compliance

Autoscaling Techniques

Different techniques exist in the literature that addresses the problem of autoscaling. As a result of the wide diversity of these techniques, which sometimes combine two or more methods, it is a challenge to find a proper classification of autoscaling techniques [44, 65]. However, these techniques could be divided into two categories: *reactive* and *proactive*. In outline, the reactive approach reacts to real-time system changes such as an increase in incoming workload while a proactive approach relies on historical access patterns of a system to anticipate future needs so as to acquire or release resources in advance. Each of these approaches has its own merits and demerits [23]. Under both categories, the most widely used autoscaling techniques range from threshold-based policies, reinforcement learning, queuing theory, control theory to time series analysis.

- **Reinforcement Learning (RL)** is a machine learning approach that enables learning through interactions between an agent and the environment. It operates on the basis of punishment and reward, biasing agents towards actions that yield the maximum reward. This technique is efficient when used against slowly-changing conditions. Therefore, it cannot be applied to real applications that usually suffer from sudden traffic bursts. RL has been applied successfully across a range of domains supporting the automated control and allocation of resources [66–70].

- **Queuing Theory** can also be applied to the design of elasticity controllers. It refers to the mathematical study of the waiting lines or

queues, taking into account the waiting time, service time, or arrival rate. Queuing theory imposes hard assumptions that may not be valid for real, complex systems. They are intended for stationary scenarios, thus models need to be recalculated when conditions of the application change. There are many works [71–74] that apply Queuing theory to the design of elasticity controllers. For example, Ali-Eldin et al. [73] model a Cloud service using queuing theory. Using that model they build two adaptive proactive controllers that estimate the future load on a service.

- **Control Theory**: Controllers that use control theory employ a model of the application, hence the performance of such controllers depends on the application model. Setting the gain parameters can be a difficult task. Previous works [20, 75–77] have extensively studied applying control theory to achieve fine-grained resource allocations that conform to a given SLO. However, the offline training feature of the existing approaches makes the deployment, model training and configuration of the elasticity controller difficult.

- **Time Series Analysis:** In time series techniques, a given performance metric is sampled periodically at fixed intervals and analysed to make future predictions. Typically these techniques are utilized to predict workload or resource usage and to derive a suitable scaling action plan. For example, Gmach et al. [78] used a fourier transform-based scheme to perform offline extraction of long-term cyclic workload patterns. CloudScale [79] and PRESS [80] perform long-term cyclic pattern extraction and resource demand prediction to scale up. Time series analysis is a purely proactive approach, whereas threshold-based policies (used in Amazon [3] and RightScale [62]) is a reactive approach. In contrast, reinforcement learning, queuing theory and control theory could be used with both proactive and reactive approaches.

Elasticity Controllers

Elastic provisioning is usually conducted automatically by elasticity controllers. The overall goal of the elastic controller is to guarantee that the allocated resources match the current demand as closely as possible. To support elasticity, a system is typically assumed to be scalable [44], i.e.,

its capacity to serve workload be proportional to the number of service instances deployed in the system. Moreover, the hosting platform needs to have enough resources to allocate whenever requested. The illusion of unlimited resources in the Cloud is central to the idea of provisioning services elastically. The changing demands of an application can be handled by dynamically scaling the system to satisfy the application demands. There are two ways of scaling in a Cloud environment: vertical scaling (scaling up or down) and horizontal scaling (scaling out or in). Vertical scaling refers to adding/removing more resources (compute, memory, storage or both) to/from each individual node to keep the performance at desired levels [63, 74, 81, 82]. Horizontal scaling refers to adding/removing nodes, which may be located at different locations. Often, load balancers are used to distribute the load among the nodes. Horizontal scaling is the most widely implemented method. Most Cloud providers such as Amazon [3] and Azure [83] and many academic works [19, 21, 23, 24, 60] use this method. Some works have also used a combination (hybrid) of horizontal and vertical scaling. For instance, Dutta et al. [84] proposed a scaling framework that combines vertical (adding resources to existing VM instances) and horizontal scaling (adding new VM instances) to ensure that the application is scaled in a way that optimizes both resource utilization and scaling-related reconfiguration costs.

Elasticity Controllers in Practice

Most of the elasticity controllers available in public Cloud services and used nowadays in production systems use threshold-based policies (rely on simple if-then threshold-based triggers). Examples of such systems include Kubernetes [85], Amazon Auto Scaling (AAS) [3], Rightscale [62], and Google Compute Engine Autoscaling [4]. The wide adoption of this approach is mainly due to its simplicity in practice, as it does not require pre-training or expertise to get it up and running. Policy-based approaches are suitable for small-scale systems in which adding/removing a VM when a threshold is reached (e.g., CPU utilization) is sufficient to maintain the desired SLO. For larger systems, it might be non-trivial for users to set the thresholds and the correct number of VMs to add/remove. Another limitation of these rule-based approaches is that they are reactive, hence they only scale resources after SLO violations have been detected. Therefore, end-users may experience performance degradation until the extra resources become available to

fix the SLO violations.

Scryer [61] is Netflix's predictive auto-scaling engine. It allows Netflix to provision the right number of instances needed to handle the traffic of their customers. Unlike systems such as AAS, Scryer predicts what the needs will be prior to the time of need and provisions the instances based on those predictions. However, its genesis was triggered more by its relatively predictable traffic patterns, which is not always true in a dynamic environment such as the Cloud.

Research on Elasticity Controllers

Most of the elasticity controllers that go beyond simple threshold-based policies, require a model of the target system to be able to reason about the status of the system and decide on control actions needed to improve the system. The system model is typically trained offline using historical data, and the controller is tuned manually using expert knowledge of the expected workload patterns and service behavior.

Work in this area focuses on developing advanced models and novel approaches for elasticity control such as ElastMan [20], SCADS Director [19], scaling HDFS [18], ProRenata [23], and Hubbub-scale [24]. Although achieving very good results, most of these controllers ignore the practical aspects of their solution which might explain the slow adoption of such controllers in production systems. For example, SCADS Director [19] is tailored for a specific storage service with pre-requisites that are not common in storage systems (fine-grained monitoring and migration of storage buckets). ElastMan [20], uses two controllers in order to efficiently handle diurnal and spiky workloads but it requires offline manual training of both controllers. Lim et al. [18] on scaling Hadoop Distributed File System (HDFS) adopt CPU utilization, which highly correlates request latency, for scaling, but it relies on the data migration API integrated into HDFS. ProRenaTa [23] minimizes the SLO violation during scaling by combining both proactive and reactive control approaches, but it requires a specific prediction algorithm and the control model needs to be trained offline. Hubbub-Scale [24] and Augment Scaling [86] argue that platform interference can mislead an elasticity controller during its decision making. However, the interference measurement needs the access of many low level metrics, e.g. cache counters, of the platform.

On the other hand, we focus on the research of the practical aspects of an elasticity controller. Our proposal relies only on the most generic and obtainable metrics from the system and alleviates the burden of applying an elasticity controller in production. Specifically, the auto-training feature of our solution makes its deployment, model training and configuration effortless. Furthermore, a generic and extendable prediction model is integrated to provide workload prediction for various workload patterns.

Feedback versus Feedforward Control

In computing systems, a controller [87] or an autonomic manager [88] is a software component that regulates the non-functional properties (performance metrics) of a target system. Non-functional properties are properties of the system such as the response time or CPU utilization. From the controller's perspective, these performance metrics are the *system output*. The regulation is achieved by monitoring the target system through a monitoring interface and adapting the system's configurations, such as the number of servers, accordingly through a control interface (*control input*). Controllers can be classified into feedback or feedforward controllers depending on what is being monitored.

In feedback control, the system's output (e.g., response time) is monitored. The controller calculates the control error by comparing the system's current output to the desired value set by the system administrators. Depending on the amount and sign of the control error, the controller changes the control input (e.g., number of servers to add or remove) in order to reduce the control error. The main advantage of feedback control is that the controller can tolerate noise and disturbance such as unexpected changes in the behaviour of the system or its operating environment. Disadvantages include oscillation, overshooting, and possible instability if the controller is not properly designed. Due to the nonlinearity of most systems, feedback controllers are approximated around linear regions called the operating region. Feedback controllers work properly only in the operating region they were designed for.

In feedforward control, the system's output is not monitored. Instead, the feedforward controller relies on a model of the system that is used to calculate the system's output based on the current system state. For example, given the current request rate and the number of servers, the system model is used to calculate the corresponding response time and act accordingly

to meet the desired response time. The advantages of feedforward control include being faster than feedback control in reaching the optimum point and avoiding oscillations and overshoot.

The major drawback of feedforward control is that it is sensitive to unexpected disturbances that are not accounted for (modelled) in the system model. Addressing this issue may result in a relatively complex system model, compared to feedback control, that tries to capture all possible states of the modelled system. Another approach is to apply online training that continuously adapts the system model in order to reflect changes in the physical system.

Workload Prediction

In order to achieve elasticity, it is important to know when and how to scale resources assigned to applications. Although horizontal scaling provides a larger-scale resource, it may take several minutes to boot a VM, and the new VM needs to be initialized with data. Therefore, it is desirable to predict workloads of Cloud services for better performance. This enables VMs to be spawned/removed in advance before SLO violations happen.

A significant amount of literature exists that can be applied for predicting the traffic incident on a service, i.e., [19, 23, 61, 80, 89]. In most cases, to support different workload scenarios, more than one prediction algorithms are used. To support different workload scenarios, at least more than one prediction algorithm is used. In most cases, the pattern of the workload to be predicted is defined. The most important aspect is how switching between prediction algorithms is carried out.

For instance, AGILE [90] provides online, wavelet-based medium-term (up to 2 minutes) resource demand prediction with adequate upfront time to start new application servers before performance degrades i.e. before application SLO is affected by the changing workload pattern. In addition, AGILE uses online profiling to obtain a resource pressure model for each application it controls. This model calculates the amount of resources required to maintain an application's SLO violation rate at a minimal level. AGILE derives resource pressure models for just CPU without considering other resources such as memory, network bandwidth, disk I/O, applications workload intensity etc. A multi-resource model can be built in two ways. Each resource can have a separate resource pressure model or a single resource pressure model can represent all the resources.

2.3 Intra-node resource management

The most common approach in large data centres is to assign tasks to cores by first allocating them to the least-loaded PMs (VM-to-PM placement), and then allocating these tasks to resources of each PM (e.g., cores, memory, etc.) using a single-node resource manager (intra-node resource management) [91]. Consolidating LCAs and BEAs on the same PM is the key to improving data centre utilization and operating efficiency [10, 15, 16, 91, 92]. Unfortunately, achieving this is difficult due to strict QoS requirements of LCAs – failing to meet latency targets can result in the loss of millions of dollars [16, 93]. Therefore, techniques to safely (without compromising QoS) consolidate multiple LCAs with multiple BEAs on the same PM are required for data centre operation efficiency and cost-effectiveness. Performance interference among the consolidated workloads is one of the main challenges of workload consolidation. It is mainly caused by the contention over shared hardware resources on the underlying PM. Interference on shared resources such as LLC, interconnect links and memory controllers can have a severe impact on the end-user experience [9]. Therefore, it is important to provide resource allocation techniques that manage interference between consolidated applications and improve the usage of shared PM resources.

Resource Management in Single-socket Architectures

Prior work has extensively explored architectural and system software techniques to tackle interference in workload consolidation environments. These techniques can be grouped into three broad approaches. The first approach is to simply avoid sharing resources with LCAs [10, 94–96]. This approach preserves the QoS of the LCAs but lowers the resource efficiency of the underlying system. The second approach avoids co-scheduling of applications that may interfere with each other [55, 97–100]. Although this approach improves resource utilization, it limits the options of applications that can be co-scheduled and may require 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 allocation techniques [10, 14–16, 94, 101–104]. This approach has several benefits: (1) it maximizes resource utilization and throughput, or trades off throughput vs. fairness [14, 57]; (2) it provides QoS for LCAs [10, 15, 16, 105]; (3) it protects

Shared Resource	Allocation Mechanism	Software/Hardware Allocation Tool
CPU Cores	Core Affinity	Linux's cpuset cgroups
LLC	Way Partitioning	Intel CAT [28]
Memory Bandwidth	Bandwidth Limiting	Intel MBA [28]
Memory Capacity	Capacity Division	Linux's memory cgroups
Disk Bandwidth	I/O Bandwidth Limiting	Linux's blkio cgroups
Network Bandwidth	Network Bandwidth Limiting	Linux's qdisc

Table 2.1: Examples of different shared resources on a NUMA machine, and their allocation tool/interface.

from timing channel attacks, where a malicious program can steal secure information, such as encryption keys, by sharing the LLC [101, 106].

Resource allocation mechanisms

With regards to resource partitioning in single-socket architectures, several resource allocation mechanisms were developed to allow lightweight partitioning of shared resources among consolidated applications to safeguard the SLO of LCAs from the BEAs interference. Table 2.1 provides examples of shared resources and corresponding allocation mechanisms. Each co-located application can be allocated some fraction of the shared resource using existing hardware/software allocation tools. However, determining the optimal allocation of these shared resources is challenging [16].

In contrast to the allocation mechanisms shown in table 2.1 that partition a single resource, some mechanisms such as thread packing [107], clock modulation [108], and CPU scheduling [92] restrict the general resource usage of an application. With thread packing, applications' threads are dynamically packed into fewer cores for power capping or resource partitioning. Thread packing can be used with multi-threaded applications without requiring any code or binary modifications. However, it cannot be applied to single-threaded applications to partition memory bandwidth.

Clock modulation, also known as duty cycle modulation, enables system software to specify the percentage of duty cycles. The corresponding CPU then skips a similar portion of clock cycles without performing any activities. Unlike thread packing, clock modulation can be applied to both single-threaded and multi-threaded applications.

Recently, CPU scheduling has been proposed to mitigate interference at microsecond timescales [92]. This approach manages interference by controlling how cores are allocated to applications. Fried et al. [92] has shown that CPU scheduling can achieve both strict performance isolation and high CPU utilization for co-located applications. However, CPU scheduling imposes new requirements on applications, such as the need to adopt a custom runtime for scheduling and the need for applications (especially LCAs) to expose their internal concurrency [92].

Since the main focus of this thesis is the allocation and arbitration of memory bandwidth, we briefly highlight memory bandwidth partitioning techniques. Memory bandwidth partitioning techniques have been extensively studied to address memory bandwidth contention between the consolidated workloads [8, 13, 109–111]. Thread packing [107], CPU scheduling [92], and clock modulation [108] are software mechanisms that have been widely-used to partition memory bandwidth on commodity servers [13]. More recently, Intel released Memory Bandwidth Allocation (MBA) as part of the RDT bundle that ships with Intel Xeon Scalable server processors. MBA supports architecture-level memory bandwidth allocation. It provides a programmable request rate controller that controls the traffic between level two cache (L2) and LLC. Thus MBA provides approximate and indirect per-core control over memory bandwidth. More importantly, MBA effectively provides memory bandwidth partitioning without throttling any non-memory-related activities (e.g., arithmetic instructions), unlike the mentioned software-based mechanisms. For example, the Intel MBA mechanism can be used to provide 20% and 80% memory bandwidth to two consolidated applications.

Resource allocation frameworks

Many systems use the aforementioned partitioning mechanisms to provide strict performance isolation and high resource utilization for data center servers [10, 13–16, 56, 57, 92, 105]. The available partitioning mechanisms can be used by a higher-level resource allocation framework to implement robust policies that achieve the overall objectives of a system (such as promoting fairness, maximizing system-wide throughput, and providing QoS for LCAs).

More recent research efforts [15, 16, 92] have focused on the QoS-aware resource allocation problem where multiple LCAs and BEAs run together

(where LCAs have QoS requirements and throughput-oriented BEAs need to achieve high performance). **PARTIES** [15], **CLITE** [16] and **Caladan** [92] aim to consolidate multiple LCAs with BEAs. PARTIES presents a significant improvement over Heracles [10] which was restricted to consolidating only one LCA with multiple BEAs. PARTIES adopts a simple approach to achieving the SLO targets of multiple consolidated LCAs. It evaluates the observed performance by incrementally increasing/decreasing one resource (e.g., number of cores, memory bandwidth, memory capacity, etc) for one LCA at a time. This process continues until (hopefully) the QoS of all LCAs is met, at which point "leftover" resources are donated to the BEAs. Although PARTIES is simple and effective, it has some inherent inefficiencies in resource utilization because it ignores the performance of throughput-oriented BEAs.

CLITE proposed a multi-resource partitioning technique based on Bayesian Optimization to address this shortcoming. The goal of CLITE is to consolidate multiple LCAs with multiple BEAs while: (1) meeting the QoS requirements of all LCAs, and (2) optimizing the BEAs' performance. CLITE employs Bayesian Optimization to sample a small number of points in large configuration space to build a low-cost performance model of various resource partitioning configurations and then navigates this search space intelligently to identify near-optimal configurations.

Fried et al. [92] proposed *Caladan*, an interference-aware CPU scheduler. Caladan consists of a centralized, dedicated scheduler core and a Linux Kernel. The scheduler core collects control signals and performs resource allocation decisions, while the Linux Kernel efficiently adjusts resource allocations. The scheduler core also differentiates between high-priority, latency-critical applications (LCAs) and low-priority, best-effort applications (BEAs). Caldan exclusively relies on core allocation to manage interference. Caladan's primary requirement is that LCAs need to expose their internal concurrency to runtime, which may necessitate changes to existing code. Furthermore, Caladan is unable to manage interference across NUMA nodes [92].

Among the proposed resource allocation frameworks, some notable examples have given significant attention to the challenges behind allocating memory-related resources (e.g., cache and memory bandwidth) across colocated applications. EMBA [57] introduced a performance model to guide the use of MBA to improve system performance. CoPart [14] proposed a resource manager that dynamically partitions the LLC (using Intel's Cache Allocation Technology (CAT)) and memory bandwidth (using MBA) to

the applications. Although these approaches are designed for single-socket servers, they also treat applications as of equal priority and lack any support for QoS.

Resource Management in Multi-socket Architectures

Most PMs in contemporary data centres are multi-socket NUMA systems. In many warehouse-scale data centres, dual-socket (or even larger) machines already constitute the largest share of hosts [7]. In this section, we first describe NUMA machines and their advantages. Second, we explain the challenges introduced by NUMA and the techniques used to address these challenges. Then, we conclude this chapter with the techniques for QoS-aware resource allocation on multi-socket systems.

Organization of NUMA Machines

Parallel architectures with non-uniform memory access (NUMA) have emerged as the norm in high-end servers. In a NUMA system, CPUs and memory are organized as a set of interconnected nodes, where each node typically comprises one or more multi-core CPUs as well as one or more memory controllers. Each memory controller provides both local and remote threads access to a partition of the physical address space. The non-uniform memory access nature stems from this organization, since the memory access bandwidth and latency depends on which node the accessing thread runs and which node the target physical page resides on. Figure 2.2 and 2.3 shows an example of a NUMA system. The key idea behind NUMA architectures is to have multiple memory controllers which increases the available bandwidth to the DRAM. Consequently, multiple memory controllers can be accessed simultaneously, allowing a much higher aggregated bandwidth. In a NUMA machine, the maximum available bandwidth is the sum of the maximum bandwidth of each memory controller.

NUMA nodes are connected by interconnect links, hence the memory accesses to remote memory are made via the interconnect links and incur a higher latency. The interconnect technology differs between processors models. For instance, Intel processors use the QuickPath Interconnect (QPI) technology or the recent Ultra Path Interconnect (UPI), while AMD processors use the HyperTransport (HT) technology or the recent Infinity Fabric (IF) technology [112]. Both technologies are "packet-based" i.e., data that is

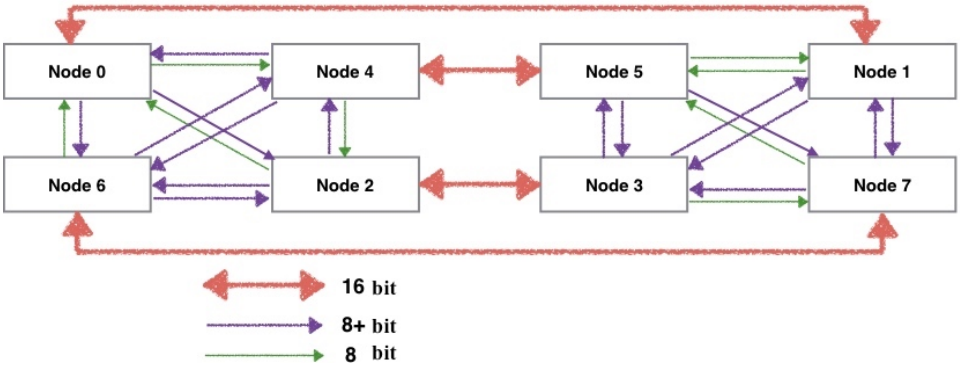


Figure 2.2: Asymmetric NUMA machine, AMD Opteron 6168 Processors with eight nodes, each hosting six cores. The width of interconnect links varies, some links are 16 bit wide (e.g., between 0 and 1), some are 8 bit wide and others are 8+ bit wide.

transmitted via the links (HT or QPI) is split into multiple packets of fixed size. Each packet includes a header that indicates the source and destination of the packet. Where the processors are not directly connected, packets can be routed and forwarded by processors to reach their destination. The interconnect links and the cache coherence protocol provide the illusion of a single DRAM connection shared by all processors, which immensely simplifies the programming of multi-threaded applications.

Memory and Thread Placement in NUMA

Exploiting NUMA architectures efficiently is notoriously challenging for programmers. Locality and resource contention issues have been identified as the two main problems to tackle [113–115]. Locality means that, for a thread to perform well, the data it accesses should be as close as possible. Better locality is expected to improve latency in two ways: it avoids remote wire delays and most importantly, decreases congestion on the interconnect links.

Furthermore, resource contention can also affect the effective performance and, sometimes, contradict the above expectations. If too many threads use some shared resources at the same time, performance degradation is to be expected. The interconnect links, the LLC and the memory controllers are examples of resources that can suffer from contention. Therefore, when the effects of congestion become significant, an appropriate

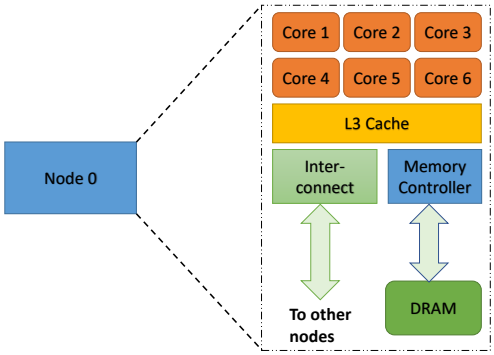


Figure 2.3: A NUMA node. It contains a set of cores and an associated set of locally connected memory.

strategy should limit the number of remote memory accesses as much as possible and to minimize contention on shared resources.

As shown in Figure 2.3, the cores of the same NUMA node typically share a last level (L3) cache, the memory controller, and the interconnect links. Assuming that all the cores of a NUMA node are performing very memory-intensive computations, the bottleneck easily becomes the shared local memory subsystem. Prior work has shown that memory controller contention and interconnect congestion are key factors hurting performance [9]. Another problem that may significantly impact the performance of parallel programs on a NUMA system is load imbalance across memory controllers [11].

Multiple techniques have been developed to avoid these issues. Most of these techniques try to improve thread placement or memory placement automatically [9, 11, 12, 112, 116, 117]. When deploying an application on a NUMA system, a number of complex questions arise, from how many threads, to where to place threads and pages. As NUMA systems increase their prominence, these problems receive increasing attention from the research community.

Regarding the problem of parallelism tuning, different techniques have been proposed, mostly in the context of SMP architectures, both for standalone and co-scheduled environments [118]. More recently, proposals like SCALO [119] and NuCore [112] have addressed this problem in the context of asymmetric NUMA systems.

Thread placement involves placing threads close to the data they are accessing. Migrating a thread is costly and has many undesirable side effects (e.g., data that were in cache have to be fetched again from memory). Thread migration also has to be carefully controlled to avoid creating CPU imbalance (e.g., overloading a CPU while leaving another CPU idle). Thread placement and scheduling is tightly coupled to the problem of page placement. While efficient solutions to thread placement/scheduling need to take into account where data is located, many systems (e.g. [112, 116]), do not control data placement. Hence, they rely on the user/programmer/OS to decide where to place pages of each application. Other solutions control both thread and page placement holistically. Examples include autonuma [117], kMAF [113], Asymsched [11], among others. Alternatively, some proposals focus exclusively on the problem of page placement, like Carrefour [12].

The page placement strategies can be grouped into two categories: *locality-oriented* page placement and *bandwidth-oriented* page placement.

Locality-oriented page placement. Locality-oriented page placement aims at minimizing memory latency. This approach departs from the assumption that, for a thread to perform well, the data it accesses should be as close as possible. For many applications, the best option is to place pages on the same nodes as the threads that are accessing such pages. The default allocation policy is dependent on the operating system. For example, on Linux, data is allocated on the node from which it is first accessed. This is referred to as the *first-touch (default) policy*. The rationale behind this heuristic is that, if a thread allocates data and continues to use it on the same node, then data is likely to be accessed locally in the future. The *libnuma* library [120] can be used to force data to be allocated on a particular node.

Memory allocation is typically static, i.e., once the data has been allocated on a node, it stays on the node. This might not work well when threads migrate between nodes or when data is accessed from multiple nodes. It works best when memory accesses to allocated data is predictable and

thread placement is known a priori. Memory migration can be used to move data between nodes in a NUMA system after allocation. The main aim of memory migration is to reduce the latency of memory access by moving data close to the processor where the process accessing that data is running. Memory migration is a dynamic technique (the same data can be migrated several times depending on the application). Since moving data from node to node incurs an overhead, memory migration is unsuitable for data that are frequently accessed from different nodes. In Linux, system calls such as *move_pages* and *mbind* can be used to migrate pages from a node to another node.

Bandwidth-oriented page placement. Bandwidth-driven page placement aims at maximizing the memory throughput of bandwidth-intensive applications. The fundamental insight of this placement is that, for bandwidth-intensive applications, their throughput improves if memory accesses are distributed across all memories, proportionally to the bandwidth of each memory. The OS can perform memory interleaving (spreading memory on multiple NUMA nodes) automatically. In Linux, the *numactl* tool can be used to this purpose. It is also possible to interleave memory of specific data. Apart from improving memory throughput, memory interleaving also aims at reducing memory controller contention and interconnect congestion. Memory interleaving can be enforced statically (by placing data uniformly on nodes during allocation) or dynamically (by periodically migrating data to balance load between nodes).

Carrefour [12] designed and implemented an algorithm that integrates a number of well-known mechanisms to place applications' memory to avoid traffic hotspots and prevent congestion in memory controllers and on interconnect links. These mechanisms include: *page co-location*, *page interleaving*, and *page replication*. **Page co-location** is about re-locating the physical page to the same memory node as the thread that accesses it. It is useful for pages that are accessed by a single thread or by threads co-located on the same memory node. **Page interleaving** entails uniformly distributing physical pages across memory nodes. Interleaving comes in handy when we have an imbalance on memory controllers and interconnect links, and when pages are accessed by multiple threads. Interleaving allocation policy is typically provided by operating systems, but only with the option to enable or disable it globally for the entire application. **Page replication**

refers to placing a copy of a page on multiple memory nodes. Replication helps alleviate traffic hotspots by distributing the load across memory controllers. Additionally, it eliminates remote accesses on replicated pages. Unfortunately, replication has costs. Since we keep multiple copies of the same page, we must synchronize their contents. Page table synchronization is another potential source of overhead. Lastly, replication also increases the memory footprint.

Linux provides several extensions [117, 121] to improve data access locality in NUMA systems. For instance, **AutoNUMA** [117] migrates threads closer to the memory they are accessing and/or migrates application data to memory closer to the threads that reference it, thereby implementing locality-driven optimization. Page fault statistics are used to determine memory accesses. A daemon periodically unmaps pages. Page faults occur when these pages are accessed, allowing AutoNUMA to compute statistics on threads and page locations. When a page induces two consecutive page faults from the same node, the page is migrated to the node. Unfortunately, AutoNUMA does not take into account workloads with data sharing i.e., if a page is accessed from multiple nodes, then it is either migrated constantly between nodes (which creates unnecessary overhead) or it is ignored by the algorithm.

Linux also provides an option to uniformly interleave part of address space across memory nodes, but the decision when to invoke the interleaving is left to the programmer or the administrator. When the application is memory-intensive, a common strategy is to uniformly interleave pages across the set of worker nodes, i.e., the nodes on which the application threads run. This strategy is based on the rationale that, for a large class of memory-intensive applications, bandwidth – rather than access latency – is the main bottleneck. Therefore, interleaving pages across nodes provides threads with a higher aggregate memory bandwidth [11]. Hereafter, let us call this strategy *uniform-workers*. This is the essential approach of recently proposed runtime libraries for NUMA systems (e.g., [11, 12] as well as the recommended or default option for prominent database systems (e.g., [122–124]). As previous research has shown, an unsuitable page placement can impact the performance of memory-intensive applications by up to a factor of 3 [12].

Windows and Solaris use the same allocation policy called the *"home node"* policy [125, 126]. A "home node" is attributed to each application by the kernel. The "home node" serves two main purposes: (i) when an

application makes a memory allocation, data is preferably allocated on the application's home node and (ii) threads of an application are preferably scheduled on the application's home node. This policy works best when threads of different applications can be clustered on different nodes without causing CPU idleness. However, home node policy can create contention for multi-threaded application, because all memory tends to be allocated on a single node.

QoS-aware Resource Allocation for Multi-socket Architectures

To the best of our knowledge, all proposed solutions to the previous QoS-aware resource allocation problem are restricted to a single-socket host. A common approach to deploy such state-of-the-art systems in a multi-socket host is to distribute applications among sockets [9, 29, 30] and consider the resources contained in each socket (CPU cores, LLC, memory) as exclusively shared among the applications in that socket. This way, each socket can be considered as an independent workload consolidation island, which can be managed by an independent instance of some QoS-aware resource allocation system. The only resources that require cross-socket allocation are external, OS-managed resources such as disk or network [15].

While simple in practice, this strict approach prevents an application running in a given socket to place data pages on remote memory nodes (from other sockets). 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 [9, 14, 29]. 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, the state-of-the-art QoS-aware resource allocation systems need to be generalized to allow cross-socket sharing of memory (as in the previous example), in order to properly utilize over-provisioned memory resources in multi-socket hosts.

Chapter 3

OnlineElastMan: Self-Trained Proactive Elasticity Manager for Cloud-based Storage Services

The pay-as-you-go pricing model and the illusion of unlimited resources in the Cloud call for the need to provision services elastically. The key insight of elastic provisioning of services is to allocate/de-allocate resources dynamically in response to the changes of the workload. It minimizes the service provisioning cost while maintaining the desired service levels as defined by SLOs. Model-predictive control [89] is often used in building such elasticity controllers that dynamically provision resources. However, they need to be trained, either online or offline, before making accurate scaling decisions. The training process involves tedious and significant amounts of work as well as some expertise, especially when the model has many dimensions and the training granularity is fine, which has been proven to be essential in order to build an accurate elasticity controller [21, 23].

In this chapter, we present OnlineElastMan, a self-trained proactive elasticity manager for Cloud-based storage services. OnlineElastMan automatically and continuously improves its provision accuracy, minimizing provisioning cost and SLO violations, as the services' workload changes as time goes by. Our experimental evaluation of OnlineElastMan using Cassandra key-value store showed that OnlineElastMan continuously

improves its provision accuracy, i.e., minimizing provisioning cost and SLO violations, under various workload patterns.

3.1 Problem Statement and Proposal Overview

There is a large body of work on elasticity controllers for the Cloud [18–24]. Most of them focus on improving the control accuracy of the controller by introducing novel control techniques and models. However, the practical issues regarding the deployment and application of the controllers are typically overlooked in literature. We examine the usefulness of an elasticity controller while deploying it in a Cloud environment. Specifically, we investigate the configuration steps for an elasticity controller before it starts provisioning services. Typically, it involves the following steps to set up an elasticity controller.

1. Acquire metrics for the elasticity controller from the provisioned application or the host platform.

2. Deploy the provisioned application in order to construct a training case for the elasticity controller.

3. Configure the provisioned application according to the deployment.

4. Configure and run a specific synthesized workload against the application.

5. Collect training data from the training case and train the control model accordingly.

6. Repeat steps 2 to 5 until the control model is fully trained before serving the real workload.

It is intuitively clear that the more metrics are considered in a control model, the more accurate it will be. However, increasing the metric dimensions of a control model comes with a significant amount of overhead during the training phase. Specifically, training a control model with only 3 dimensions results in 27 (3^3) training cases when only 3 trials/runs are conducted for each dimension. This means that step 2 to step 5 needs to be repeated 27 times to train the control model. Obviously, it is extremely time-consuming

to train a control model manually, especially when the model has many dimensions, which is needed for higher control accuracy. In this chapter, we focus on the problem of training a control model for an elasticity manager in the Cloud in an effective and seamless manner.

OnlineElastMan alleviates the training process with online training. Specifically, the control model automatically trains and evolves itself while serving the workload. After a short period of warm-up, the controller can provision the underlying application accurately. Thus, it is no longer needed to manually and repetitively reconfigure the system in order to train the control model. Furthermore, in order to make OnlineElastMan as general as possible, its input metrics are easily obtainable from the application (i.e., with little or even no instrumentation of the application). Specifically, it directly uses the information in the incoming workload and SLO status as the input metrics.

In addition, previous works [18, 23] have demonstrated that, in order to keep the SLO commitment, a storage service needs to scale up *in advance* i.e., proactively and not reactively, to tackle a workload increase since scaling a storage service involves non-negligible overhead. Thus, we have made a design choice to integrate a workload prediction module for OnlineElastMan. Again, to make it as general as possible, the workload prediction module is able to produce accurate workload predictions for various workload patterns. Specifically, it has integrated several prediction algorithms that are designed to cope with different time-series patterns. The most appropriate prediction algorithm is chosen online using a weighted majority selection algorithm.

3.2 Target System

We are targeting multi-tier web applications (the left side of Fig. 3.1). We are focusing on managing the data-tier because of its major effect on the performance of web applications, which are mostly data-centric [48]. For the data-tier, we assume horizontally scalable key-value stores due to their popularity in many large-scale web applications such as Facebook and LinkedIn. A typical key-value store provides a simple put/get interface. This simplicity enables efficient partitioning of the data among multiple servers and thus scales well to a large number of servers.

The minimum requirements to enable elasticity control of a key-value

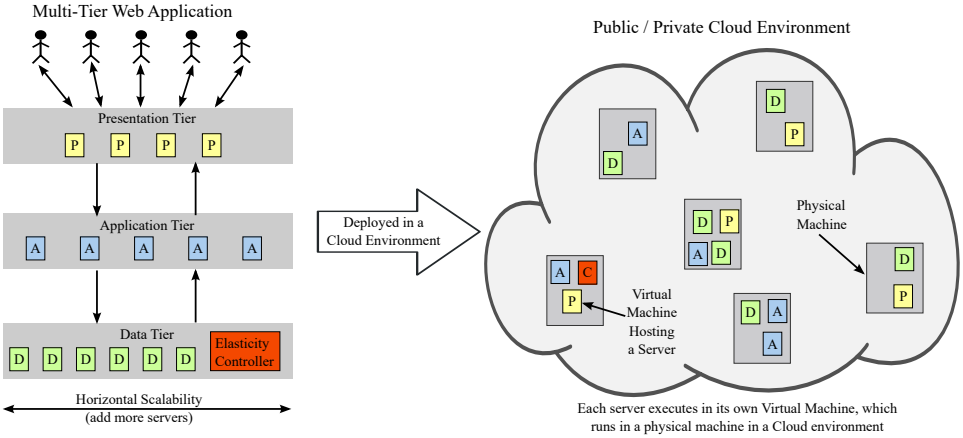


Figure 3.1: Multi-Tier Web Application with Elasticity Controller Deployed in a Cloud Environment.

store are as follows. The store must provide a monitoring interface that provides information about the current workload, and whether the service is currently meeting the SLO or not. The store must also provide an actuation interface that allows horizontal scalability by adding or removing servers. As storage is a stateful service, actuation must be combined with a rebalance operation, which redistributes the data among the new set of servers. Many stores, such as Voldemort [51] and Cassandra [53], provide rebalancing tools.

We target applications running in the Cloud (right side of Fig. 3.1). We assume that each service instance runs on its own VM; each physical machine hosts multiple VMs. The Cloud environment hosts multiple applications (not shown in the figure). Such an environment complicates the control problem mainly due to the fact that VMs compete for the shared resources. This environmental noise makes it difficult to model and predict the performance of that VMs (possibly running different services/applications) [127, 128].

Cassandra

We have chosen Cassandra as our targeted underlying distributed storage system. Cassandra [36] is open-sourced under Apache licence. It is a distributed storage system which is highly available and scalable. It stores

column-structured data records and provides the following key features:

- **Distributed and decentralized architecture:** Cassandra is organized in a peer-to-peer fashion. Specifically, each node performs the same functionality in a Cassandra cluster. However, each node manages a different namespace, which is decided by the hash function in the DHT (distributed hash table). Comparing to Master-slave, the design of Cassandra avoids a single point of failure and maximizes its scalability.

- **Horizontal scalability:** The peer-to-peer structure enables Cassandra to scale linearly. The consistent hashing implemented in Cassandra allows it to swiftly and efficiently locate a queried data record. Virtual node techniques are applied to balance the load on each Cassandra node.

- **Tunable data consistency level:** Cassandra provides tunable data consistency options, which is realized through different combinations of read and write APIs. These APIs use *ALL, EACH_QUORUM, QUORUM, LOCAL_QUORUM, ONE, TWO, LOCAL_ONE, ANY, SERIAL, LOCAL_SERIAL* to describe read/write calls. For example, the *ALL* option means Cassandra reads/writes all the replicas before returning to clients. A detailed specification of each read/write option can be found in [129].

- **An SQL like query tools - CQL:** The common access interface in Cassandra is exposed using Cassandra Query Language (CQL). CQL is similar to SQL in its semantics. For example, a query to get a record whose id equals 100 results in the same statement in both CQL and SQL (*SELECT * FROM USER_TABLE WHERE ID*= 100). It reduces the learning curve for developers to use CQLs and get started with Cassandra.

3.3 OnlineElastMan's Architecture and Design

In this section, we present the architecture and design of OnlineElastMan. It relies on four main components, which Figure 3.2 depicts. At the lowest level, a monitoring component is expected to be provided as part of

the distributed storage service (Cassandra, in our case study). This component collects and provides through some API, relevant workload metrics and information about whether the underlying service's SLO is currently met or not. Then, above this component, three main components comprise the core of OnlineElastMan: workload prediction, online model training, and elasticity controller. Figure 3.2 presents the architecture of OnlineElastMan. Components operate concurrently and communicate by message passing. Briefly, the workload prediction module takes input from the current workload and predicts the workload for the near future (the next control period). The online model training module updates the current control model by mapping and analyzing the monitored workload and the performance of the system. Then, the elasticity controller takes the predicted workload and consults the updated control model to issue scaling commands by calling the Cloud API to add or remove servers for the underlying storage system.

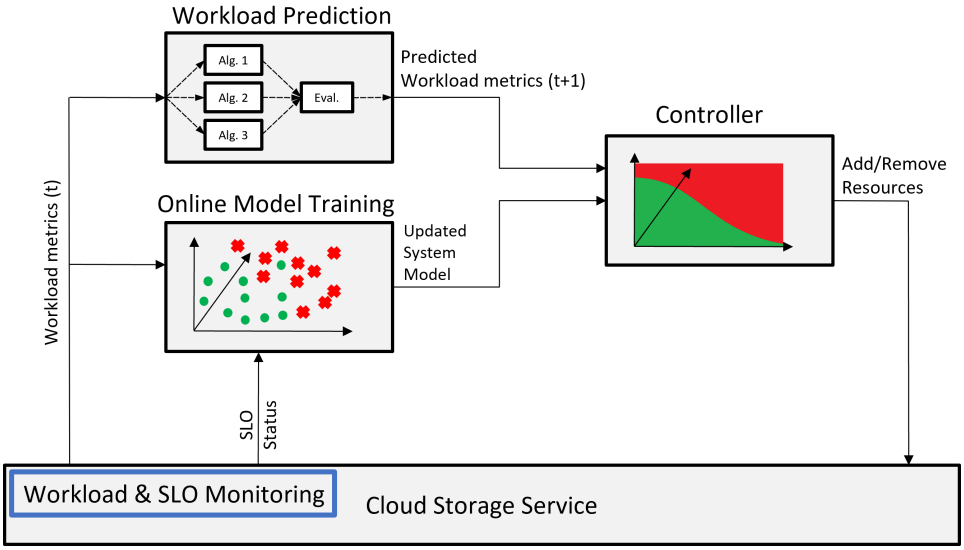


Figure 3.2: OnlineElastMan Architecture.

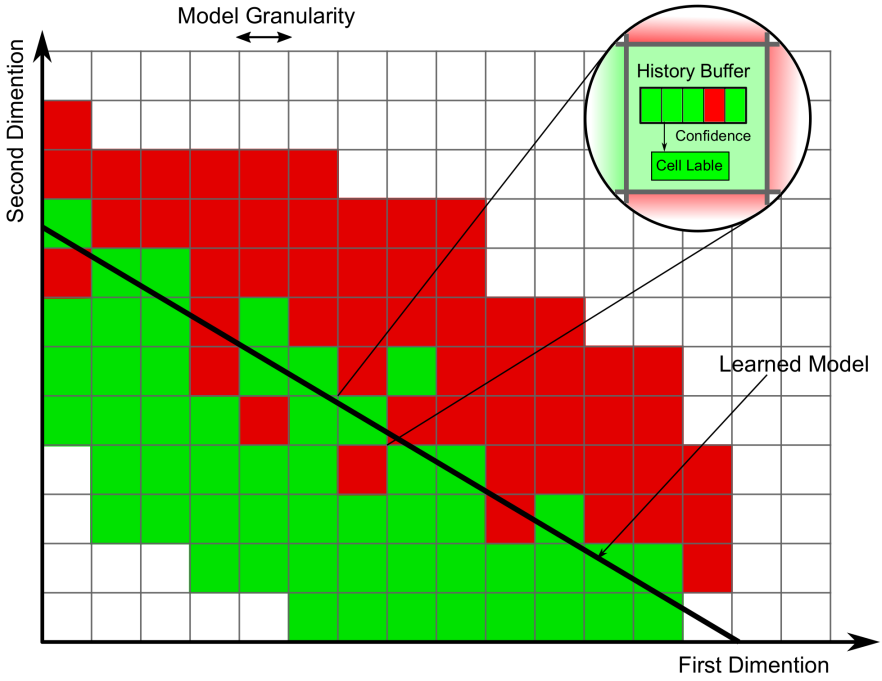


Figure 3.3: Building the SML model.

Workload and SLO Monitoring

Auto-scaling techniques require a monitoring component that gathers various metrics that reflect the real-time status of the targeted system at an appropriate time granularity (e.g., per second, per minute, per hour). It is essential to review the metrics that can be obtained from the target system and the metrics that best reflect the status of the system. To ease the configuration of the OnlineElastMan framework and to make it as general as possible, we consider the target storage system as a black box. A monitoring component is expected to be provided as part of the distributed storage service (Cassandra, in our case study). This component collects and provides, through some API, relevant workload metrics and information about

whether the underlying service's SLO is currently met or not. We expect that the workload metrics are defined to cover a set of relevant workload features whose variation has a strong impact on the resulting performance of the targeted storage system – and, therefore, on its ability to fulfill its SLOs.

For our example storage system, i.e., Cassandra, for the workload metrics, the client requests that reach a Cassandra server are processed, and three metrics are extracted:

- Read and write intensity, i.e., number of read invocations and write invocations (respectively), per unit of time (e.g., reads per second and writes per second).

- Average data size, i.e., the average data size requested in read invocations and sent in write invocations.

As for the SLO in our Cassandra example, we collect the 99^{th} percentile read latency to characterize the QoS behaviour of the system. The SLO is satisfied when such value is below a pre-defined threshold. The workload metrics can be obtained by sampling the traffic passing through the entry points, e.g. proxies or load balancers, of the managed storage service(s). The percentile latency, which defines and directly reflects the QoS, is collected either from entry proxies or the storage service itself depending on the design and workflows of storage services. In Section 3.4, we provide further details on how we obtain these metrics with Cassandra, as well as other distributed storage service.

Online Model Training

It is intuitively clear that with more provisioned resources (VMs), the system is able to respond to requests with reduced latency. However, on the other hand, we would also like to provision as little VMs as possible to save the provisioning cost. Thus, the controlled latency range is always desired to be slightly under (just satisfying) the percentile latency requirement defined in the SLO to minimize the provisioning cost. We refer to this region as **optimal operational region (OOR)**, where a system is not very much over-provisioned but satisfying the SLO. To achieve this goal, an elasticity controller needs to estimate what the OOR is for a given system (at a given

period). In order to keep the system operating in the OOR while the incoming workload is dynamic, an elasticity controller needs to react to the workload changes and allocate/de-allocate VMs to the system.

OnlineElastMan devises a multi-dimensional Statistical Machine Learning (SML) model to determine/estimate OOR. This model correlates the workload metrics (workload characteristics) with the SLO metric (such as percentile latency). This SML model constitutes OnlineElastMan's control model.

In an n-dimensional space, where n is the number of workload metrics collected by OnlineElastMan, the SML model is a hyperplane that defines the boundary between two zones, where the SLO is predicted to be satisfied and violated, respectively. This hyperplane is built from a limited training set, obtained from the monitoring component. Each element in the training set corresponds to a point in the n-dimensional space of workload metrics and is labeled according to whether the service was able to meet the SLO with the workload represented by that point or not.

For instance, with 3 workload metrics (as in the Cassandra case study described so far), the hyperplane is a 2-dimensional plane, as shown in Figure 3.6. If the number of dimensions is 2, then the hyperplane is just a line (as shown in Figure 3.5).

This hyperplane allows us to estimate the OOR of a given VM in our system. Intuitively, the OOR corresponds to the points located under but close enough to the hyperplane. Furthermore, it allows us to estimate whether, with the current workload, each VM is operating within its OOR and, if not, how far that VM is from the OOR. As we detail later in the chapter, these estimates enable the Elasticity Controller of OnlineElastMan to decide how many VMs should be allocated or removed to ensure that the system operates in its OOR. Next, we start by describing the SML model used in OnlineElastMan, and then we explain how such model is built.

Multi-Dimensional Online Model

In contrast to previous proposals [19, 20, 23, 80] that rely on an offline SML model, OnlineElastMan builds the model online and continuously improves/updates the model while serving requests. The online training feature frees system administrators from the tedious offline model training procedure, which includes repetitive system configurations, system deployments, model updates, etc., before putting the controller online. Addition-

ally, the continuously evolving model in OnlineElastMan enables the system to survive with factors that are not considered in the model, e.g. platform interference [60, 130].

Specifically, the online SML model is built with the monitored parameters mentioned in Section 3.3. It classifies whether a VM is able to operate in the OOR under the current workload, which breaks down to the read/write intensity and the requested data size. Ideally, a storage node hosted in a VM can either be operating under the SLO commitment or with the SLO violation. Therefore, for a given VM, the SML model is a line that separates the plane into two regions, in which the SLO is either met or violated, as shown in Figure 3.5. Different SML models need to be built for different VM flavours[1] and different storage services hosted. While building the SML model as depicted in Figure 3.3, several configurable parameters affect the SML model accuracy. We discuss the most important ones next.

Granularity of the SML model: Since the collected data for the model can be decimal, it is impossible to analyze the data with infinite combinations. We group the collected data with a pre-defined granularity, which makes a two-dimensional plane to be separated into small squares or a three-dimensional plane to be separated into small cubes. These squares and cubes are the groups where data are accumulated and analyzed. The granularity of data groups can be configured depending on the memory limits and the precision requirements of the SML model.

Historical data buffer: For data collected and mapped to each group, we maintain a historical record for the most recent n reads and writes.

Confidence level: The historical data in each group is analyzed to define whether the workload that corresponds to the data collected in this group violates the SLO or not. To do so, we take into account a pre-defined confidence level parameter. For example, the 95% confidence level implies that 95% of all the read/write percentile latency sampled satisfy the SLO.

Update frequency: The SML model updates itself periodically with a fixed configurable rate. A higher update frequency allows the SML model to swiftly adapt to execution environment changes while a lower update frequency makes the SML model more stable and tolerate transient execution environment changes.

[1] A flavour is a VM instance type, or virtual hardware template. A flavour specifies a set of virtual machine resources such as the number of virtual CPUs, the amount of memory, and the disk space assigned to a VM instance.

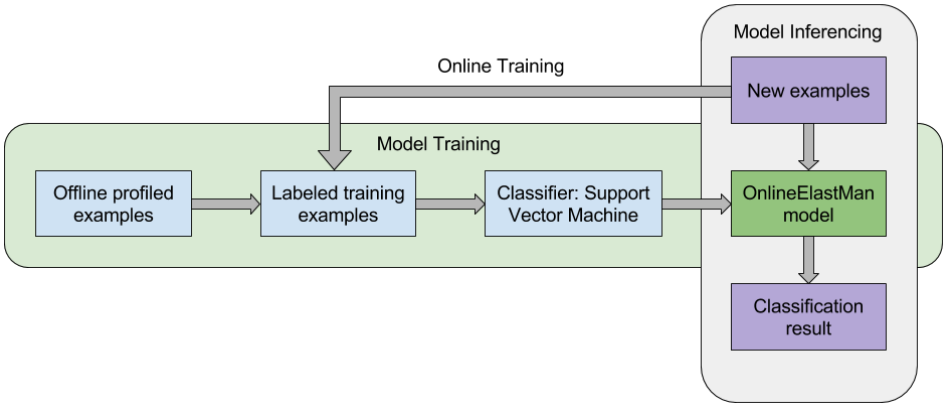


Figure 3.4: Classification using SVM.

Building the SML Model

To build the multi-dimensional SML model described in the previous section, we resort to Support Vector Machine (SVM), a supervised machine learning method. SVM have become popular classification techniques in a wide range of application domains [131]. They provide good performance even in cases of high-dimensional data and a small sets of training data. Using the *"kernel trick"*, SVM is also able to find non-linear solutions efficiently [132]. Figure 3.4 shows the flow of a classification task using SVM. Briefly, we first train the SML model offline using systematically profiled data. Then, we put the SML model online to let it evolve itself.

Next, we describe how OnlineElastMan applies SVM to build each VM's SML model. Each instance of the training set (data point) contains a class label and several features or observed variables. In our Cassandra case study, the features are read intensity, write intensity and average data size, while the class label is either 1 (SLO is met) or -1 (SLO is violated). Recall that, the goal of SVM is to produce a model based on the training set. Specifically, its goal is to find a hyperplane in an n-dimensional space (where n denotes the number of features, i.e. workload metrics). Figures 3.5 and 3.6 depict two models in systems which consider 2 and 3 features, respectively. Each model can be seen as a decision boundary that helps classify whether, for some given point in the workload space (as characterized by the n features of some particular workload) the VM is predicted to serve

the workload while safeguarding the SLO (in case the point is below the model) or not (otherwise).

To separate the two classes of data points, there are many possible hyperplanes that could be chosen. The objective of SVM is to find a hyperplane that has the maximum margin, i.e the maximum distance between data points of both classes. Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane.

More concretely, given a training set of instance-label pairs $(x_i, y_i), i = 1, ..., l$ where $x_i \in R^n$ and $y_i \in \{1, -1\}^l$, the SVM classification solves the following optimization problem [133]:

$$min_{w,b} \qquad \| w \|^2 + C \sum_i \xi_i \tag{3.1}$$

subject to:

$$\begin{aligned} y^{(i)}(w^T x^i + b) &\geq 1 - \xi_i, \quad i = 1, 2, \ldots, l \\ \xi_i &\geq 0, \quad i = 1, 2, \ldots, l \end{aligned} \tag{3.2}$$

Each training example (instance of a training set) is denoted as x, and superscript (i) denotes the (i^{th}) training example. y superscripted with (i) represents class label corresponding to the (i^{th}) training example. l denotes the size of the training set, while n denotes the number of dimensions. The parameter C in the SVM optimization problem is a positive cost factor that penalizes misclassified training sets. A larger C discourages misclassification more than a smaller C. The non-negative variables, $\xi_i \geq 0$, were introduced to enable the optimal separating hyperplane method to be generalised. The ξ_i are a measure of the misclassification errors. w is known as the weight vector. After solving the above equation, the SVM classifier predicts 1 if $w^T x + b \geq 0$ and -1 otherwise. The decision boundary is defined by the following equation:

$$w^T x + b = 0 \tag{3.3}$$

Generally, the predicted class can be calculated using the following linear discriminant function:

$$f(x) = wx + b \tag{3.4}$$

where x refers to a training example, w to the weight vector and b to the bias term. wx refers to the dot product, which calculates the sum of the products of vector components $w_i x_i$. For example, in case of training set with three features (e.g. x, y, z corresponding to reads per second, writes

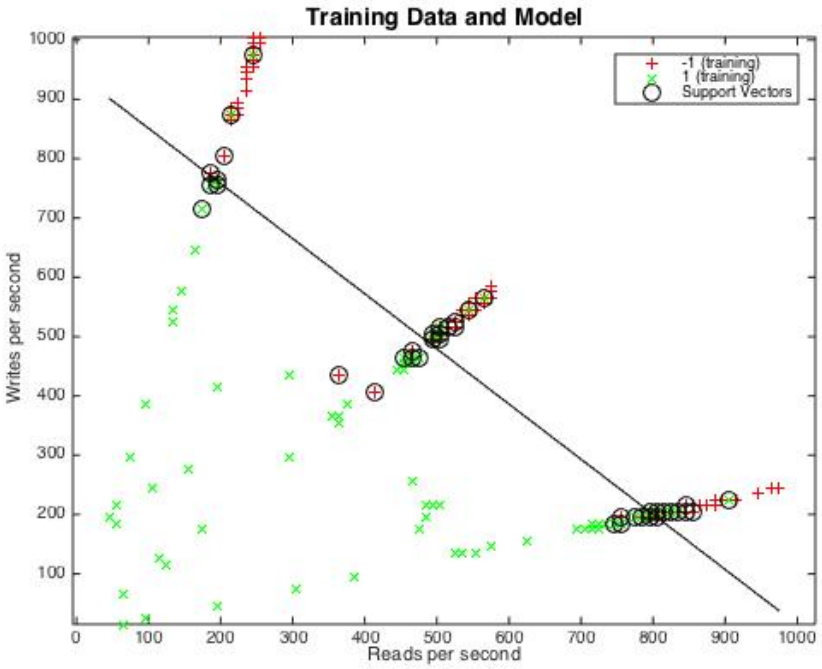


Figure 3.5: 2-dimensional SML model. Each point represent a training example in 2-dimensional space (where 2 is the number of features, i.e., reads per second and writes per second). Each data point also has a class label $(1, -1)$. The red points (-1) signifies SLO is violated, while the green points (1) signifies SLO is met. Support vectors are data points that are closer to the model line and influence the position and orientation of the line.

per second and average data size respectively), the discriminant function is simply:

$$f(x) = w_1 x + w_2 y + w_3 z + b \tag{3.5}$$

SVM provides the estimates for w_1, w_2, w_3 and b after training.

Given Equation 3.3, the SML model is a line (Figure 3.5) when only monitoring read/write request intensity in the workload or a two-dimensional plane (Figure 3.6) when another dimension, i.e., data size, is modeled. Figure 3.7 is a 2-dimensional projection of Figure 3.6, which shows that different

Figure 3.6: 3-dimensional SML model taking into account request data size.

data sizes cause different separations of the 2-dimensional model space. It indicates that data size plays an essential role to build an accurate control model for storage systems. The line/plane separation in the model represents the maximum workload that a VM can serve under the specified SLO (percentile latency).

Online model training: Using the SVM model training technique, the performance model is updated periodically at the parameterized **update frequency** using the data in the **historical data buffer** processed with the configured/pre-defined/parameterized **confidence level**.

For generality, we assume that every VM can have a significant perfor-

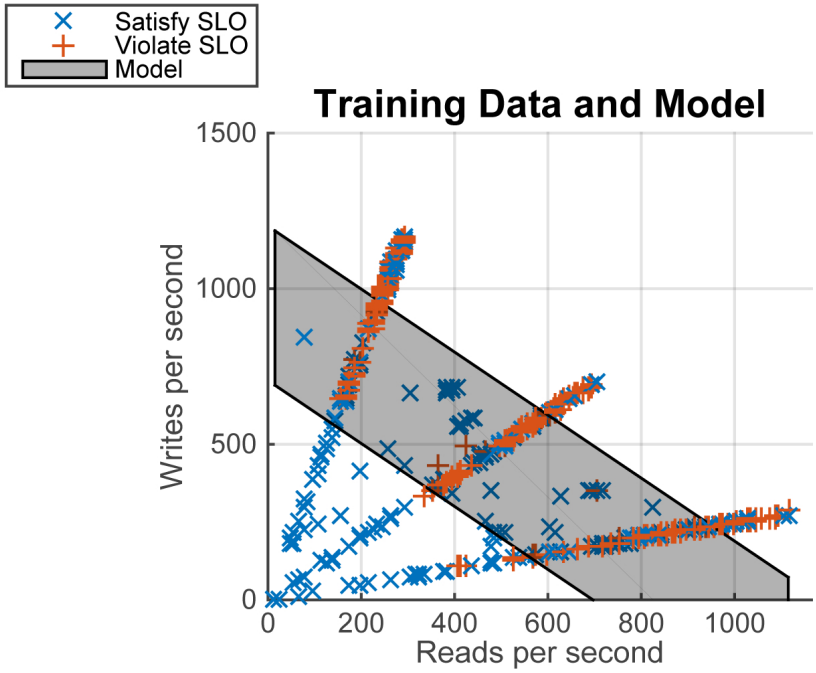


Figure 3.7: Top angle view of the SML model, where the model plane is projected to a 2 dimensional surface and the shaded area is caused by the varying data sizes.

mance difference even when they are spawned with the same flavour. This can be caused by the interference from the host platform [60, 130] or background tasks, such as data migration [23]. Thus, an individual SML model is built for each VM participating in the system. They automatically evolve and update continuously while the system is serving workload. Periodically, the updated SML models for each VM are sent to the elasticity controller module to make scaling decisions.

Elasticity Controller

An elasticity controller makes scaling decisions in configurable control periods/intervals to prevent the system from oscillations. Any scaling decision

taken by the elasticity controller is based on two main inputs. The first input corresponds to the current workload metrics of each VM x, which we denote w_x.

The second input to the scaling decision is an estimate of the capacity of each VM, c_x. The capacity of a VM is the maximum workload that it can handle under the SLO. It is inferred from the VM's SML model. Specifically, the capacity of each VM is determined by calculating the intersection point between its SML model and the line connecting the origin with the point that corresponds to the current workload (i.e., the point corresponding to read and write intensity, as well as average data size of the workload). Note that, if the current workload point is beyond (i.e., above) the capacity representation point in the model, then SLO is violated.

The responsibility of an elasticity controller is to keep the provisioned system operating while satisfying the SLO. The strictest requirement is that every VM operates while satisfying the SLO, which can be denoted by $\forall x \in N, w_x < c_x$, where N is the complete set of all participating VMs. However, this is not trivial to achieve without over-provisioning the system in case the workload distribution is not well balanced. It is challenging to balance workload in storage services with respect to each VM. This is because, as discussed in Section 2.1, storage services are stateful, i.e., usually each VM is responsible only for a part of the total data stored. Thus, a specific request can only be served by a specific set of VMs, which host the requested data. Given that different storage services have different data distributions, as well as load balancing strategies, and that OnlineElastMan is designed to be a generic framework to provision storage services elastically, we choose not to manage workload/data distribution for provisioned systems. Furthermore, managing data distribution or rebalancing among VMs is orthogonal to the design goal of OnlineElastMan. Nevertheless, OnlineElastMan provides suggestions for workload distributions to each participating VMs based on their capacity learnt from our SML models.

In order to tolerate load imbalance among VMs, OnlineElastMan introduces an optional tolerance factor α when computing scaling decisions to prevent too much over-provisioning. Specifically, a scaling up decision is issued when the SLO violation $c_x < w_x$ is observed from more than α VMs, where $\alpha \geq 0$. When $\alpha = 0$, there is no tolerance on load imbalance. The number of VMs to add is calculated individually for each VM and aggregated globally. The number of VMs (with the same flavour as c_x) that is expected to be added is given by $\frac{w_x - c_x}{c_x}$. Thus, when $\frac{w_x - c_x}{c_x} < 0$, it represents that a

VM has more capacity than the incoming workload. We aggregate results of $\frac{w_x - c_x}{c_x}$ on each VM flavour and ceil the aggregated results. When the result on a specific VM flavour is negative, we do nothing because that indicates that a scale up procedure involving that VM flavour is already ongoing. In contrast, when the result on a specific VM flavour is positive, we add the number of VMs of that flavour accordingly.

For scaling down, there is also a corresponding load imbalance tolerance factor β. β denotes the number of VMs which are over-provisioned, in each VM flavour. A scaling down procedure is triggered by first satisfying that there is no VM that violates the SLO, which gives $\forall i \in N, w_i < c_i$, where N is the complete set of all participating VMs. Then, the number of VMs to de-allocate is calculated through a similar process comparing to scaling up. The aggregated results of $\frac{w_x - c_x}{c_x}$ on each VM flavours are floored after subtracting β. Last, the corresponding number of VMs are de-allocated when the floored results are greater than zero.

When a scaling up/down decision is made, the elasticity controller interacts with Cloud API to request/release VMs. Where applicable, the elasticity controller also calls the distributed storage service API to rebalance data to the newly added VMs or to decommission the VMs that are about to be removed. Adding/removing VMs to a distributed storage service introduces a significant amount of data rebalancing load in the background. This leads to fluctuations in sensitive performance measures, such as percentile latency. Usually, the extra data rebalancing load is not long-lasting. So, this fluctuation can be filtered out in our SML model by properly setting the **confidence level** and **update frequency** of OnlineElastMan introduced in Section 3.3.

Workload Prediction

An optional but essential component of OnlineElastMan is the workload prediction module. It is always too late to make a scaling out decision when the workload has already increased since preparing VMs involves a non-negligible overhead. This is especially relevant for storage services, which require data to be migrated to the newly added VMs. Thus, there is a prediction module that facilitates OnlineElastMan to make decisions in advance.

Often, there are patterns that can be found in the workload, such as the diurnal pattern [134]. These patterns become vague when the workload is

distributed to each VM. Thus, we are not predicting the incoming workload for each VM. Rather, the workload is predicted for the whole system. Then, it is proportionally calculated for each VM based on the current workload portion that is served by the VM. Finally, instead of using the current incoming workload to make a scaling decision in the previous section, we are able to use the predicted workload as the input.

However, even predicting the workload for the whole system is not trivial since many factors contribute to the fluctuation of the workload [135]. Some workloads have a repetitive/cyclic pattern, such as diurnal patterns or seasonal patterns, while some other workloads experience exponential growth over a short period, which can be caused by market campaigns or special offers. Considering that there are no perfect predictors and different applications' workloads are distinct, no single prediction algorithm is general enough to be suitable for most workloads. Thus, we have studied and analyzed several prediction algorithms that are designed for different workload patterns, namely, **regression trees** [136] and five **ARIMA models** [137] (**first-order autoregressive**, **differenced first-order autoregressive**, **exponential smoothing**, **second-order autoregressive** and **random walk**). Then, a weighted majority algorithm (Section 10) is used to select the best prediction model. Note that as shown in Figure 3.2, for simplicity of analysis and presentation, OnlineElastMan only predicts the read and write intensities of the input workload. Next, we detail each algorithm.

Regression Trees model

Regression trees [136] do not have classes. Instead, there is a response vector y which represents the response values for each observation in variable matrix x. Regression trees predict responses to data and are considered as a variant of decision trees. They specify the form of the relationship between predictors (input variables) and response (output variable). We first build a tree using the time series data consisting of the timestamped workload metrics collected by the monitoring module, through a process known as recursive partitioning (Algorithm 1) and then fit the leaf values to the input predictors. Concretely, in our Cassandra case study, we consider timestamped read and write intensity measurements. Particularly, to predict a response, we follow the decisions in the tree from the root node all the way to a leaf node which contains the response.

Algorithm 1: Recursive Partitioning Algorithm

Data: A set of N data points, x_i, i = 1,...,n
Result: A regression tree
if *termination criterion exist* **then**
  Generate Leaf Node and allocate it a Given Value;
  Return Leaf Node;
else
  Identify Best Splitting test $s*$;
  Generate node t with $s*$;
  Left_branch(t) = $RecursivePartitioning(< x_i, y_i >: x_i = s*)$;
  Right_branch(t) = $RecursivePartitioning(< x_i, y_i >: x_i \neq s*)$;
  Return Node t;
end

ARIMA

The autoregressive moving average (ARMA) is one of the most widely used approaches to time series forecasting. ARMA model is convenient for modelling time series data which is stationary. In our case, the time series data consist of timestamped read and write intensity measurements of input workload. Stationary means that statistics calculated on the time series are consistent over time, like the mean or the variance of the observations. In order to handle non-stationary time series data, the ARMA model adopts a differencing component to help deal with both stationary and non-stationary data. This class of models with a differencing component is referred to as the autoregressive integrated moving average (ARIMA) model. Specifically, the ARIMA model is made up of an autoregressive (AR) component of lagged observations, a moving average (MA) of past errors and a differencing component (I) needed to make a time series to be stationary. The MA component is impacted by past and current errors while the AR component shows the recent observations as a function of past observations [138].

In general, an ARIMA model is parameterized as ARIMA(p,d,q), where **p** is the number of autoregressive terms (order of AR) i.e., the number of lag observations in the model, **d** is the number of differences needed for stationarity, and **q** is the number of lagged forecast errors in the prediction equation (order of MA). The following equation represents a time series

expressed in terms of the AR(p) model:

$$Y^{'}(t) = \mu + \alpha_1 Y(t-1) + \alpha_2 Y(t-2) + \ldots + \alpha_n Y(t-p) \tag{3.6}$$

Equation 3.7 represents a time series expressed in terms of moving averages of white noise and error terms. In the equations, μ is a constant, $0 < \alpha \leq 1$, $0 < \beta \leq 1$, and ϵ is white noise.

$$Y^{'}(t) = \mu + \beta_1 \epsilon(t-1) + \beta_2 \epsilon(t-2) + \ldots + \beta_n \epsilon(t-p) \tag{3.7}$$

In OnlineElastMan, apart from regression tree, we have integrated five ARIMA models, which are the first-order autoregressive ($ARIMA(1,0,0)$), the differenced first-order autoregressive ($ARIMA(1,1,0)$), the simple exponential smoothing ($ARIMA(0,1,1)$), the second-order autoregressive ($ARIMA(2,0,0)$) and the random walk ($ARIMA(0,1,0)$). In our view, they can capture almost all the common workload patterns. For example, the first-order autoregressive model performs well when the workload is stationary and autocorrelated while, for non-stationary workload, a random walk model might be suitable. Then, the challenge is to detect and select the most appropriate prediction model during runtime.

The Weighted Majority Algorithm

A Weighted Majority Algorithm (WMA) is implemented to select the best prediction model during runtime. WMA is a machine learning algorithm that is used to build a combined algorithm from a pool of algorithms [139]. WMA assumes that one of the known algorithms in the pool will perform well under the current workload without prior knowledge about the accuracy of the algorithms. WMA have many variations suited for different scenarios including infinite loops, shifting targets and randomized predictions. We present WMA in Algorithm 2. Specifically, for each workload metric being predicted, the algorithm maintains a list of weights w_1,...,w_n. Each weight estimates the prediction quality of a given algorithm. At each iteration of WMA, the weight of each prediction algorithm is updated according to the difference between the value that the algorithm predicted (for the workload metric under consideration) and the real value. More precisely, the weights of those algorithms whose mispredictions was higher than a predefined tolerance interval are penalized by multiplying their weights with a fixed penalty factor m ($0 \leq m < 1$). The prediction result from the most

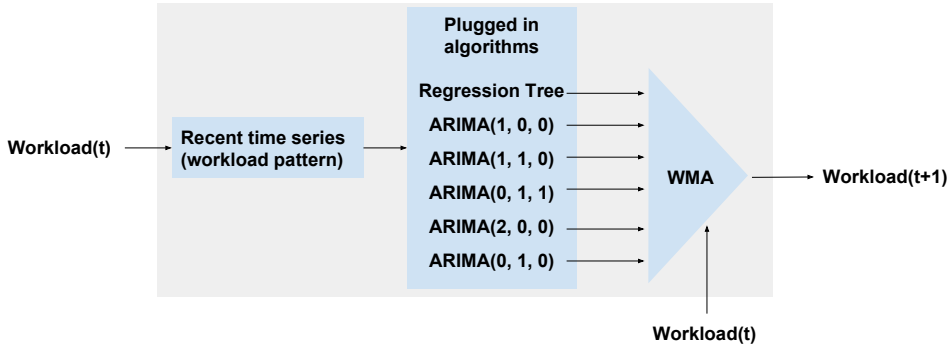


Figure 3.8: Architecture of the workload prediction module.

weighted algorithm (i.e., algorithm with the highest weight), is selected and returned.

Algorithm 2: The Weighted Majority Algorithm

1. 1. Initialize the weights $w_1, ..., w_n$ of all the prediction algorithms to a positive weight (1).
2. 2. Return the prediction result of the prediction algorithm with the highest weight.
3. 3. Compare the predicted value with the real value, penalize the prediction algorithms, which missed the prediction more than a predefined tolerance interval n, by multiplying their weights with a fixed penalty factor m $(0 \leq m < 1)$.
4. 4. Wait until next prediction interval and go to 2.

The prediction module of OnlineElastMan is shown in Figure 3.8. Additional prediction algorithms can be plugged into the prediction module to handle more workload patterns.

Putting Everything Together

OnlineElastMan operates according to the flowchart illustrated in Figure 3.9. The incoming workload is fed to two modules, the prediction module and the online training module. The prediction module utilizes the current workload composition to predict the workload in the next control period

using the algorithm described in Section 3.3. The online training module records the current workload composition and samples the SLO metric (such as service latency) under the current workload. Then, the module trains the performance model with the **update frequency**. The actuation is calculated based on the predicted workload for the next control period using the updated performance model according to the algorithm explained in Section 3.3. Finally, the actuation is carried out on the Cloud platform that hosts the storage service.

3.4 Evaluation

We evaluate OnlineElastMan from two angles. First, we study the accuracy of the prediction module, which consists of six prediction algorithms. It directly influences the provision accuracy of OnlineElastMan since it is an essential input parameter for the performance model. Then, we present the evaluation results of OnlineElastMan when it dynamically provisions a Cassandra cluster with the application of the online multi-dimensional performance model.

Our evaluation is conducted in a private cloud, which runs OpenStack software stack. Our experiments are conducted on VMs with two virtual cores (2.40GHz), 4GB RAM and 40GB disk size. They are spawned to host storage services or benchmark clients. OnlineElastMan is configured separately on one of the VMs. The overview of the evaluation setup is presented in Figure 3.10.

Evaluation Environment

Underlying Storage System

Cassandra (version 2.0.9) is deployed as the underlying storage system and provisioned by OnlineElastMan. Cassandra is chosen because of its popularity, as it is used as scalable backend storage by many companies, e.g. Facebook. Briefly, Cassandra is a distributed replicated database, which is organized with distributed hash tables. Since a Cassandra cluster is organized in a peer-to-peer fashion, it achieves linear scalability. Minimum instrumentation is introduced to Cassandra's read and write path, as shown in Figure 3.11. The instrumented library samples and stores the service latency of requests in its repository. OnlineElastMan's data collector

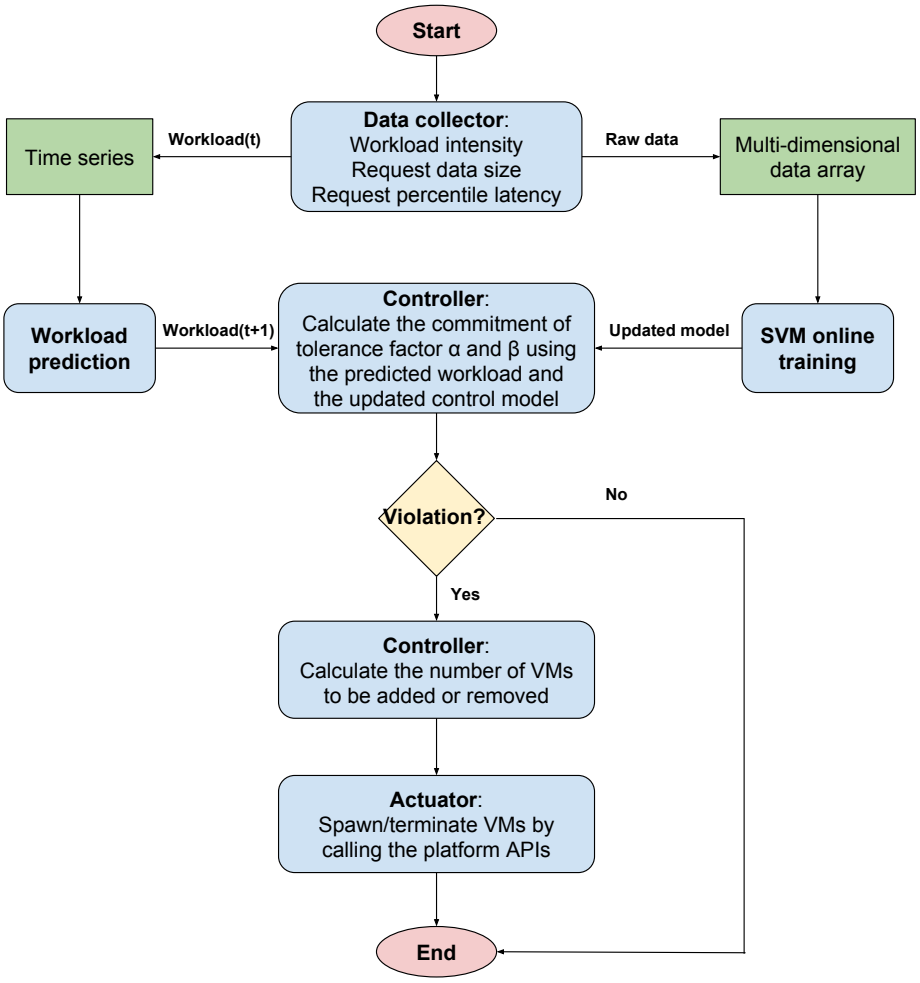


Figure 3.9: Control flow of OnlineElastMan.

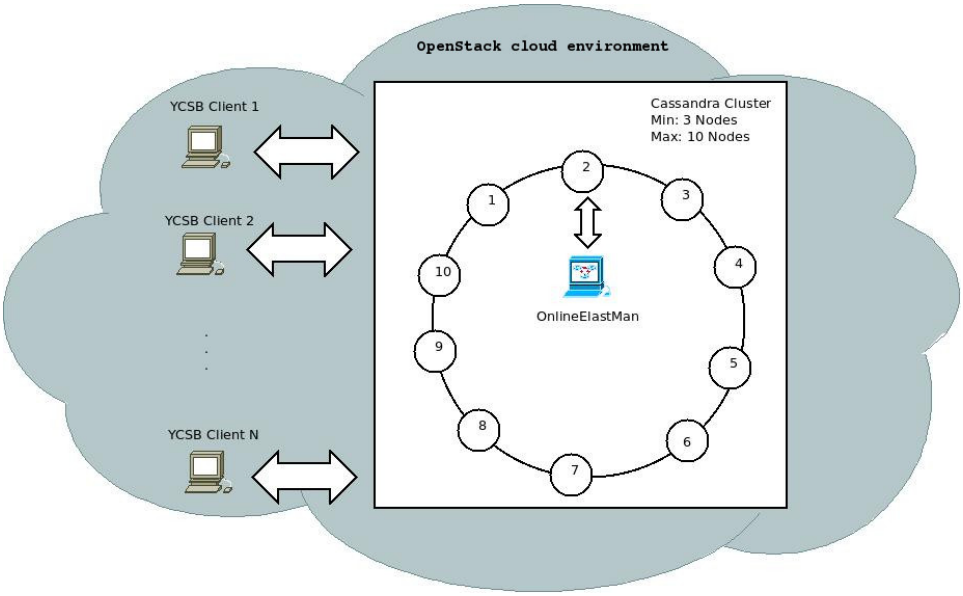


Figure 3.10: Different number of YCSB clients are used to generate workload with different intensity. OnlineElastMan resizes the Cassandra cluster according to the workload.

component periodically (after every 5 minutes in our experiments) pulls collected access latencies from the repository on each Cassandra node. The collected request samples from each Cassandra node are used by the prediction module and the online training module of OnlineElastMan, as shown in Figure 3.9. The Cassandra rebalance API is called to redistribute data when adding/removing Cassandra nodes.

Workload Benchmark

We adopt YCSB (Yahoo! Cloud System Benchmark) (version 0.1.4) to generate workload for our Cassandra cluster. We choose YCSB because of its flexibility to synthesize various workload patterns, including the varying read/write request intensity and the size of the data propagated. Specifically, we configure YCSB clients with the parameters shown in Table 3.1. In order to generate a stable workload for Cassandra, a fixed request rate (1200 req/s) is set to each YCSB client hosted on a separate VM. We vary

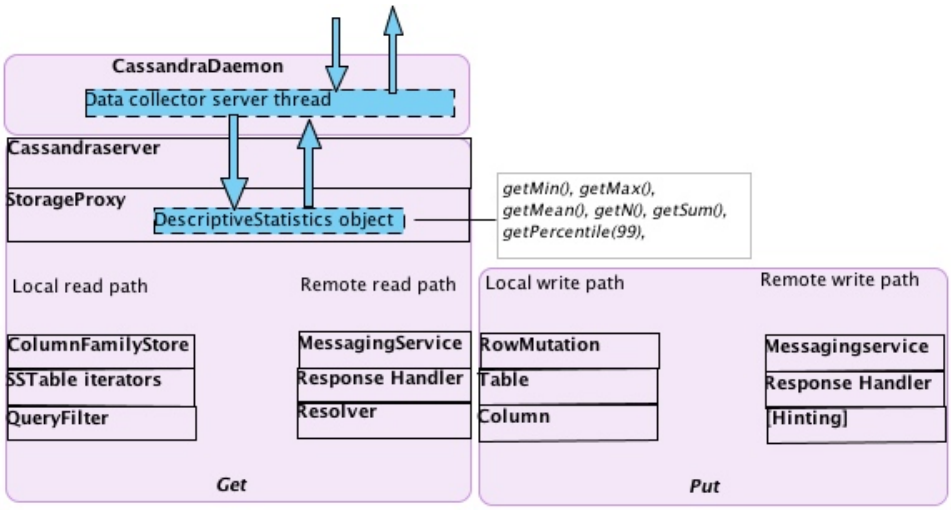


Figure 3.11: Cassandra instrumentation for collecting request latencies.

Table 3.1: YCSB configuration

Number of Threads	16
Request Distribution	uniform
Record Count	100000
Read Proportion	varied (0.0 - 1.0)
Update Proportion	varied (0.0 - 1.0)
Data size	varied (1 - 20) KB
Replication Factor	3
Consistency Level	level ONE

the total amount of workload generated by adding or removing VMs that host YCSB clients.

Multi-dimensional Performance Model

Our performance model is automatically trained when the input workload varies. OnlineElastMan takes input from the monitored parameters as specified in Section 3.3. Specifically, the workload features, including read and write request intensity and request data size, and the corresponding

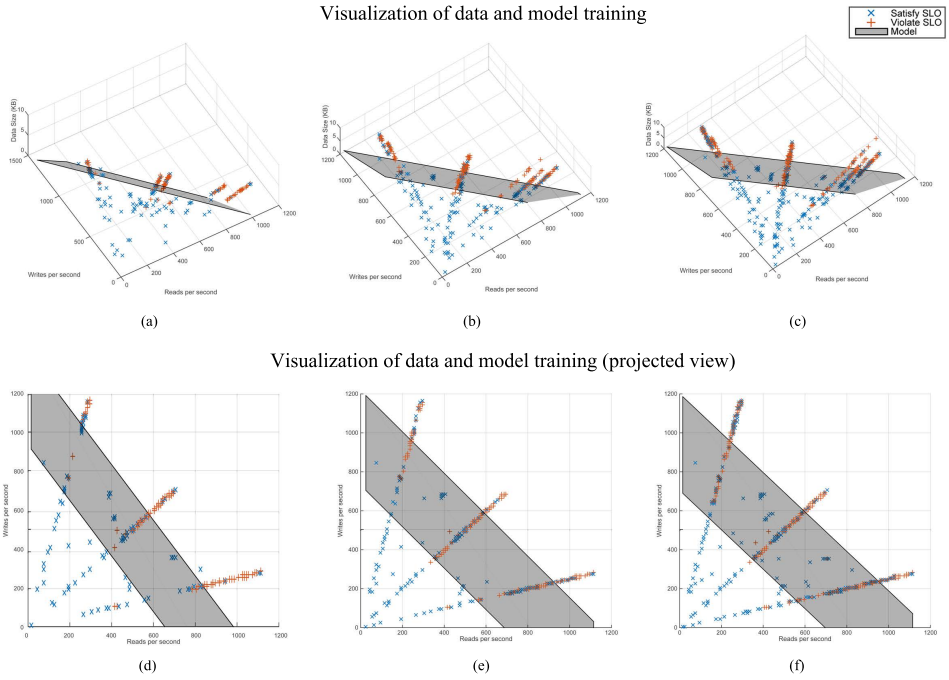


Figure 3.12: The training illustration of the 3 dimensional performance model. (a), (b), and (c) are ordered by the length of training period. (d), (e), and (f) are the visualization of (a), (b) and (c) with data size dimension projected on the other 2 dimensions.

service latency, obtained from Cassandra instrumentation, are associated to train the model. Details on model training is presented in Section 3.3.

In practice, the model starts empty and needs to get trained online automatically for some time. This is because the model is application and platform-specific. Thus, it needs a warm-up training phase. According to our experiments, it takes approximately 20 to 30 minutes to train a performance model from scratch. After the warm-up, the model can be used to facilitate the decision making process of the elasticity controller while serving the workload.

Figure 3.12 depicts the model built and used in our evaluation. It consists of three input parameters or dimensions, i.e., read/write request intensity

and the data size. The controlled parameter is the 99^{th} percentile read latency, which is set to be $35ms$ in our case. As shown in the figure, with more training data, the model (the shaded surface) evolves itself to a more accurate state. Practically, the performance model is dynamic and evolves while serving the workload. So, it can automatically evolve to a more accurate model that reflects the changes in the operating environment and the provisioned storage system. To be specific, the model adapts to unknown factors, such as application interference or platform maintenance, gradually using updated training data. A more accurate model leads to better provision accuracy when the elasticity controller consults it.

In our experiments, we found out that the rate at which the model evolves affects the accuracy of the decisions made by the controller. The **confidence level** and **update frequency** (as introduced in Section 3.3) dictates how fast the model evolves. Ideally, we should have enough confidence about the status (violate SLO or satisfy SLO) of a data point before its status changes. Setting the confidence level low and the update frequency high may result in the model oscillating (unstable model), while the opposite settings of these two parameters may delay the evolution of the model. In our experiments, we set the confidence level as 0.5, i.e., if 50% of all read and write latency queue samples satisfy the SLO, then the corresponding data point satisfies SLO and vice versa. The update frequency is set to 5 minutes. For applications that have distinct phases of operations, to prevent frequent retraining, one can maintain a set of models and dynamically select the best model for the current input pattern [90]. This idea is left for future work.

Evaluation of Workload Prediction

We evaluate the prediction accuracy of the workload prediction module using a synthesized workload generated by YCSB. We have synthesized workload with different shapes by increasing and decreasing the total request intensity. Figure 3.13 presents the actual workload generated and the workload predicted by our prediction module. In addition, the choice of the dominant prediction algorithm proposed by the weighted majority algorithm is also shown in the figure. As a result, our prediction module is able to achieve as low as 4.60% on the Mean Absolute Percentage Error for such a dynamic workload pattern.

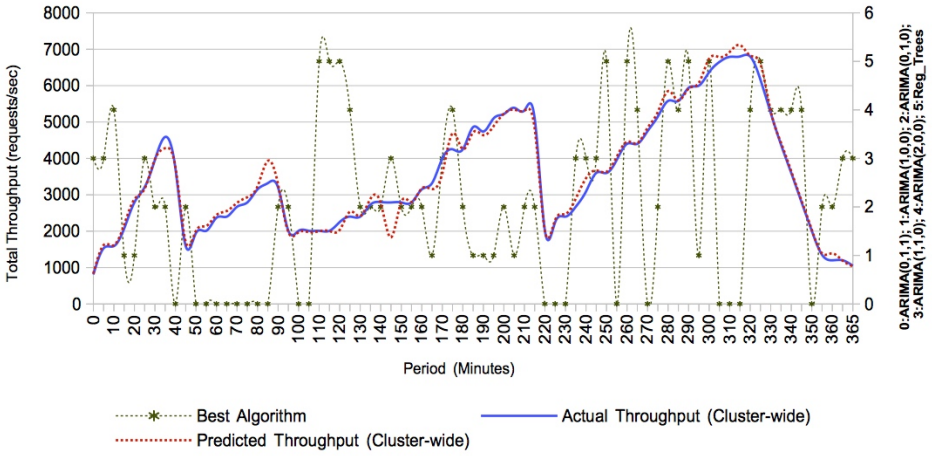


Figure 3.13: Workload prediction: the actual workload V.S. the predicted workload.

Evaluation of OnlineElastMan over Cassandra

We set the goal of OnlineElastMan to keep the 99^{th} percentile of read latency to be $35ms$ as stated in the SLO. The evaluation is conducted with control period set to be 5 minutes. Even though the workload of YCSB is configured to be uniform in our case, we still observe a non-trivial difference in the amount of workload served from different Cassandra storage VMs. To make a trade-off between the uneven workload served on each VM and preventing over-provisioning, we set the tolerance factors $\alpha = 1$ and $\beta = 0.5$.

As shown in Figure 3.14, we start the experiment with 3 Cassandra VMs. From 0 to 40 minutes, the multi-dimensional performance model is trained and warmed up. The elasticity controller starts to function after 40 minutes. From 40 to 90 minutes, the workload increases gradually. It is observable that from 40 to 70 minutes, the system is over-provisioned, as the percentile latency is far below the SLO boundary, as shown in Figure 3.15. This is because the elasticity controller is set to operate with a minimum number of 3 VMs, which corresponds to the replication factor of Cassandra. As the workload increases, the elasticity controller gradually adds two VMs from $80th$ minute on. Then, the workload experiences a sharp decrease from 90

minute, but the controller maintains a minimum of 3 Cassandra VMs. We continue to evaluate the performance of OnlineElastMan with another two rounds of workload increase and decrease with different scales (shown from 150 to 220 minutes and from 220 to 360 minutes). The evaluation indicates that OnlineElastMan is able to keep the 99^{th} percentile latency commitment most of the time. On the other hand, we observe a small amount of SLO violations under the provisioning of OnlineElastMan. It is because of the tolerance factor α and β, which allows us to tolerate some imbalance of workload distribution to Cassandra nodes.

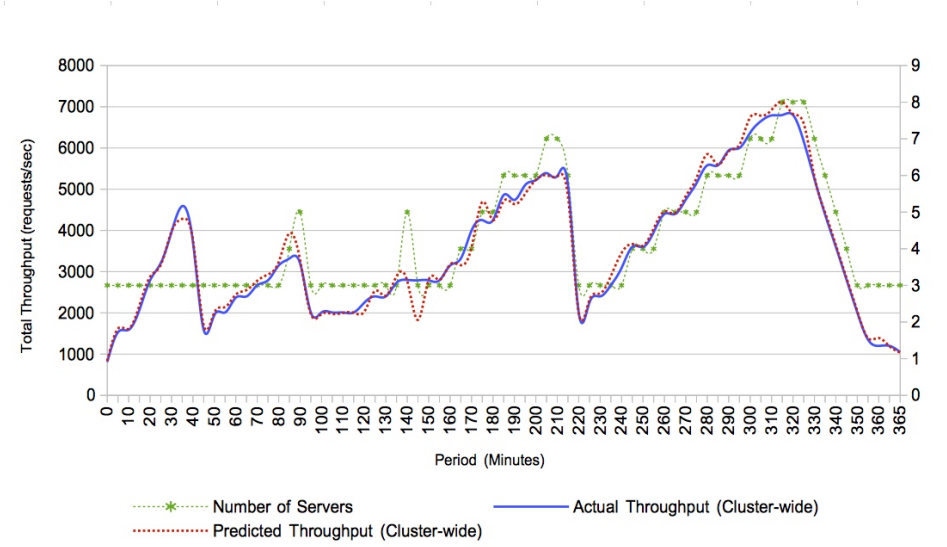


Figure 3.14: VMs allocated according to the predicted workload and the updated control model.

3.5 Related work

The majority of elasticity controllers offered in public Cloud services and deployed in production systems today use threshold-based policies (rely on simple if-then threshold-based triggers). Examples of such systems include Kubernetes [85], Amazon Auto Scaling (AAS) [3], Rightscale [62], and Google Compute Engine Autoscaling [4]. Threshold-based approaches are

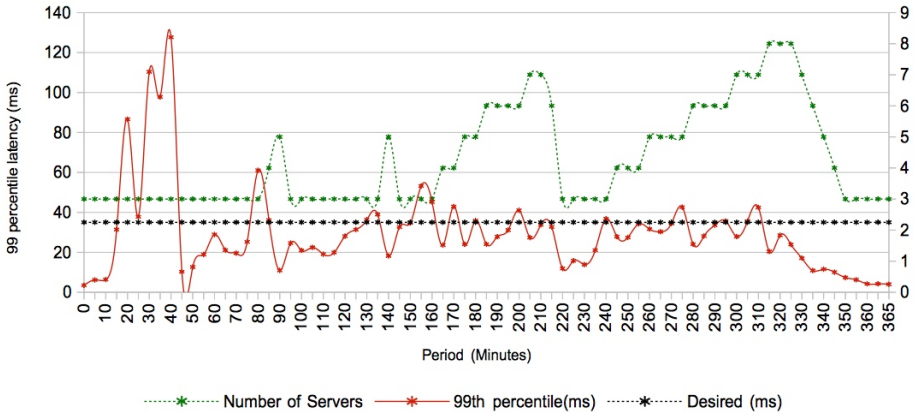


Figure 3.15: The aggregated 99^{th} percentile latency from all Cassandra VMs with the allocation of VMs indicated by OnlineElastMan under the dynamic workload.

appropriate for small-scale systems where adding/removing a VM when a threshold (e.g., CPU utilization) is reached is sufficient to maintain the desired SLO. However, setting the thresholds and the correct number of VMs to add/remove may be difficult for users on larger systems. Additionally, threshold-based approaches are reactive, hence they only scale resources after SLO violations have been detected. Therefore, end-users may experience performance degradation until the extra resources become available to fix the SLO violations.

Most of the elasticity controllers that go beyond simple threshold-based policies require a model of the target system to be able to reason about the status of the system and decide on control actions needed to improve the system. The system model is typically trained offline using historical data, and the controller is tuned manually using expert knowledge of the expected workload patterns and service behavior. Recent work in this area mostly focuses on developing advanced models and novel approaches for elasticy control, such as ElastMan [20], SCADS Director [19], scaling HDFS [18], ProRenata [23], and Hubbub-scale [24]. Although achieving very good results, most of these controllers ignore the practical aspects of the solution, which slowed down the adoption of such controllers in production systems.

In contrast, OnlineElastMan focuses on the research of the practical aspects of an elasticity controller. It relies only on the most generic and obtainable metrics from the system and alleviates the burden of applying an elasticity controller in production. Specifically, the auto-training feature of OnlineElastMan makes its deployment, model training and configuration effortless. Furthermore, a generic and extendable prediction model is integrated to provide workload prediction for various workload patterns.

Regarding workload prediction, a significant amount of literature exists that can be applied for predicting the traffic incident on a service, i.e., [19, 23, 61, 80, 89]. To support different workload scenarios, at least more than one prediction algorithm is used. In most cases, the pattern of the workload to be predicted is defined, which is not in our case. The most important aspect is how switching is carried out among the prediction algorithms. However, this is not clearly addressed in most of these previous works. We therefore propose a simple weighted majority algorithm to handle this.

In OnlineElastMan, since we do not know the pattern of our workload, we have chosen some of the types of ARIMA models that are commonly encountered. For a time series that is stationary and autocorrelated, a possible model for it is a first-order autoregressive model. On the other hand, if the time series is not stationary, the simplest possible model for it is a random walk model. However, if the errors of a random walk model are autocorrelated, perhaps a differenced first-order autoregressive model may be more suitable. Nau Robert [137] presents a detailed explanation of these models.

3.6 Summary

In this chapter, we presented the design and implementation of OnlineElastMan, which is an "out-of-the-box" elasticity controller for distributed storage systems. It includes a self-training multi-dimensional performance model to alleviate model training efforts and provide better provision accuracy, a self-tuning prediction module to adjust the prediction to various workload patterns, and an elasticity controller to calculate and carry out the scaling decisions by analyzing the inputs from the performance model and the prediction module. The evaluation results of OnlineElastMan on Cassandra show that OnlineElastMan is able to provision a Cassandra cluster efficiently and effectively with respect to the percentile latency SLO.

Chapter 4

BWAP: Bandwidth-Aware Page Placement

Page placement is a critical problem for memory-intensive applications running on a shared-memory multiprocessor with a NUMA architecture. As we show in this chapter, state-of-the-art page placement proposals for NUMA architectures fail to maximize memory throughput for many workloads. This is especially true with modern NUMA systems, characterized by asymmetric bandwidths and latencies, and sensitive to memory contention and interconnect congestion phenomena.

In this chapter, we present BWAP, a novel page placement mechanism based on asymmetric weighted page interleaving. BWAP combines an application-agnostic, but architecture-aware analytical performance model of the target NUMA system with on-line iterative tuning of page distribution for a given memory-intensive application. Our experimental evaluation with representative memory-intensive workloads shows that BWAP performs up to 66% better than state-of-the-art techniques. These gains are particularly relevant when multiple co-located applications run in disjoint partitions of a large NUMA machine or when applications do not scale up to the total number of cores. Some passages in this chapter have been quoted verbatim from [140].

4.1 Problem Statement and Proposal Overview

Parallel architectures with non-uniform memory access (NUMA) are emerging as the norm in high-end servers. In a NUMA system, CPUs and memory are organized as a set of interconnected nodes, where each node typically comprises one or more multi-core CPUs as well as one or more memory controllers. The non-uniform memory access nature stems from this organization, since the memory access bandwidth and latency depends on the node where the accessing thread runs and on the node where the target page resides.

When one deploys a parallel application on a NUMA system, its threads allocate and access pages that need to be physically mapped to the available NUMA nodes. This raises a crucial question: *where should each page be mapped for optimal performance?* When the application is memory-intensive, a common strategy is to uniformly interleave pages across the set of *worker* nodes, i.e., the nodes on which the application threads run. This strategy is based on the rationale that, for a large class of memory-intensive applications, bandwidth – rather than access latency – is the main bottleneck. Therefore, interleaving pages across nodes provides threads with a higher aggregate memory bandwidth [11]. Hereafter, let us call this strategy *uniform-workers*. This is the essential approach of recently proposed runtime libraries for NUMA systems (e.g., [11, 12]), as well as the recommended or default option for prominent database systems (e.g., [122–124]).

This work starts by questioning the effectiveness of the *uniform-workers* strategy in contemporary NUMA systems. Two key characteristics of *uniform-workers* seem to be at odds with the systems it aims to optimize. First, *uniform-workers* places pages at symmetric ratios across worker nodes, while the bandwidth (and latency) of contemporary NUMA architectures is typically asymmetric across nodes [11]. Second, there are important scenarios where an application's threads are clustered together on a *subset* of NUMA nodes – notably, when the application is deployed in a given node partition of a co-scheduled system [141], or when the application does not scale beyond a subset of the available cores [112, 142]. If the remaining memory nodes are idle or underused (e.g., by CPU-intensive applications), *uniform-workers* will neglect an important portion of bandwidth. Hence, as we show in Section 4.2, it is unsurprising that the memory bandwidth attained by *uniform-workers* is considerably suboptimal for memory-intensive applications.

To overcome these inefficiencies, we propose BWAP, a novel bandwidth-aware page placement for memory-intensive applications on NUMA systems. In contrast to *uniform-workers*, BWAP takes the asymmetric bandwidths of every node into account to determine and enforce an optimized application-specific *weighted interleaving*. Our proposal is inspired by recent research for hybrid memory systems [38–40]. These works have shown that, when a CPU (or GPU [39]) is served by different memory technologies (such as NVRAM or DRAM) with differing bandwidths, an optimal placement is one that (proportionally) place fewer pages at the lower-bandwidth memories.

Still, applying the same principle to the context of NUMA systems is far from trivial. While previous bandwidth-aware proposals for hybrid memory systems relied on the premise that a given memory node provides the same bandwidth to every core, that is no longer true in a NUMA system. The same NUMA memory node may be accessible through different bandwidths by different threads, depending on each thread's location within the NUMA topology. This implies that optimizing page interleaving from the perspective of a given worker node (as done by the recent proposals for hybrid systems [38–40]) will not always yield the best overall performance. Instead, the optimization problem needs to consider a complex $W \times N$ bandwidth matrix, where W and N denote the number of worker nodes and total nodes, respectively. Furthermore, this bandwidth matrix is particularly hard to determine accurately, since it is sensitive to interconnect congestion and local-remote contention on memory controllers phenomena which, in turn, depend on the memory demand patterns of the deployed application(s). Hence, optimal placements are eminently application-specific.

Putting it all together, an efficient page placement for asymmetric NUMA systems requires tuning N weights, taking into account complex phenomena that depend both on the underlying NUMA architecture and the application(s) itself. A naive approach is to search through the N-dimensional space of possible weight distributions and measure the performance of each run to find the optimal placement. This is often impractical for NUMA systems of 4 nodes and beyond, since it easily falls in the range of hours or days to find an optimized distribution of per-node weights for a given application. An alternative approach is to model the usage of the memory system bandwidth and analytically determine the optimal page placement. However, to the best of our knowledge, the most successful analytical models of memory throughput are limited to single-node scenarios [143].

BWAP tames aforementioned complexity by combining techniques from

the two extremes of the solution space. In a first stage, BWAP builds a memory bandwidth model of the target system. From this model, BWAP calculates the optimal weight distribution that maximizes the performance of a reference *bandwidth-intensive* application. The key insight behind BWAP is that, after analytically determining that *canonical* weight distribution, that distribution can be adjusted to fit the target application by applying a scalar coefficient on each weight. In other words, BWAP reduces what in theory is an N-dimensional optimization problem to the one-dimensional problem of finding an appropriate scaling coefficient that best fits the application. To achieve this, the second stage of BWAP relies on an iterative technique, which, when the application starts, places its pages according to the canonical weight distribution; then, on-the-fly, it uses an incremental page migration scheme that adjusts the weight distribution until a new (local) optimum is found.

BWAP is implemented as an extension to Linux *libnuma*. It enriches the original interface with a *bandwidth-interleaved* policy option that automatically determines memory nodes to place the application pages on, and the per-node weights to balance the page interleaving across the NUMA nodes. BWAP is readily available and can be used transparently by any application, with no changes to the OS kernel.

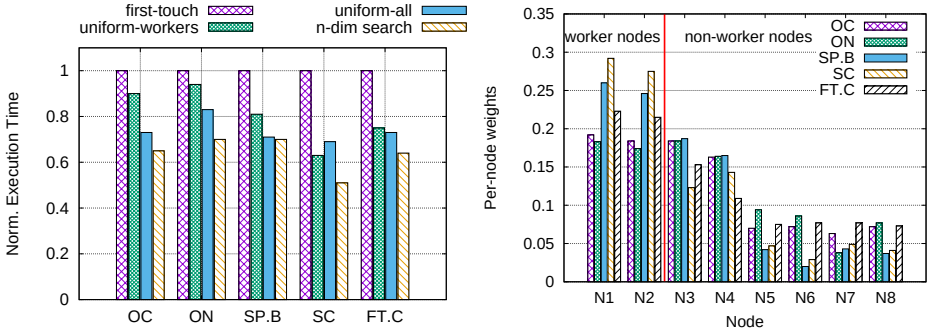
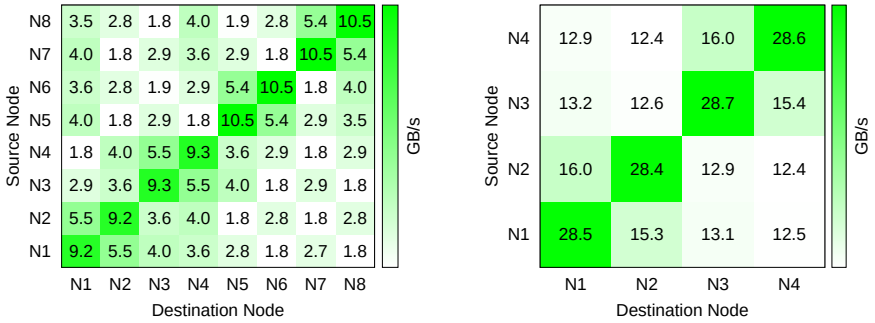


Figure 4.1: **(a):** (a) Performance of popular page placement schemes vs the placement found via n-dimensional search for Ocean_cp (OC), Ocean_ncp (ON), SP.B, Streamcluster (SC) and FT.C [1] (2 worker nodes, 8 threads each). (b) The corresponding per-node weights as found by our n-dimensional search.



(a) Machine A (b) Machine B

Figure 4.2: Node-to-node maximum bandwidths (GB/s) in machines A (8-node AMD Opteron) and B (4-node Intel Xeon).

4.2 Motivation

To lay the groundwork, we selected different memory-intensive applications from the PARSEC [33], SPLASH [34], and NAS [1] benchmark suites and experimentally studied how different page placement strategies affect their performance. We used an 8-node NUMA machine with the asymmetric interconnect topology depicted in Figure 4.2a, on which each application ran stand-alone on 2 worker nodes. A larger number of architectures and baselines is evaluated in Section 4.4.

For each application, we measure its performance when its pages are mapped with Linux default *first-touch*, the common practice *uniform-workers*, and a *uniform-all* variant that uniformly interleaves pages across all nodes (both workers and non-workers). We also performed a long offline search in which we experimentally tested the performance of a large sample of weight distributions. The search used the hill climbing technique to explore the 8-dimensional space of possible solutions. Each point in the search space assigns to each memory node in the machine a weight that determines the portion of pages that will be placed at that node. The starting point in the search was *uniform-workers*. Each search covered approximately 180 iterations, taking more than 15 hours to complete for each application. For each application, the search identified a number of slightly different configurations that achieved performance within less than 3% from optimum. Thus, the values discussed next are averages over a selection of the top-10

best performing distributions for each application.

Figure 4.1a presents the performance of the baseline policies normalized with respect to that of hill climbing. The results suggest that, while *uniform-workers* and *uniform-all* considerably improve performance over Linux's default policy, they do not take full advantage of the bandwidth of the underlying memory architecture in our NUMA system. To understand why, we have studied the actual weight distributions obtained by hill climbing, as depicted in Figure 4.1b and drawn the following three main observations, which guided us towards the design of BWAP.

Observation 1: Pages are placed across all nodes, not just worker nodes. In many modern architectures, even applications that have moderate single-thread memory demands can easily saturate the local memory controller when multiple threads share the same node. This issue is further exacerbated if the application threads span across multiple NUMA nodes, since a fraction of accesses to pages will now be remote, thus limited by the bandwidth of the interconnect. These results suggest that, for applications with high memory demands, page placement should not be restricted to the worker nodes; instead, the available (even if limited) bandwidth of non-worker nodes should be harnessed by placing on these nodes a carefully selected fraction of the application's pages.

Observation 2: Pages are interleaved unevenly across nodes, with relevant cross-application variations. Every weight distribution obtained by hill climbing is highly asymmetric, with nodes with lower memory access throughput receiving fewer pages. This clearly reflects the inherent asymmetry of the underlying NUMA topology [11]. However, when we compare the best weight distributions found for different applications, we observe significant differences between applications, which we can explain by two main factors. On the one hand, the complex contention effects, both at the interconnect and memory controller [9], depend on the actual memory demand that the application places on each memory node. On the other hand, while memory bandwidth is the dominant bottleneck of some applications, others are more sensitive to memory latency. The former benefit from exploiting the bandwidth of remote nodes to its full extent; the latter call for approaches that, while spreading some pages remotely for increased bandwidth, retain most pages locally for the sake of latency.

Observation 3: If one considers worker nodes and non-worker nodes separately, proportional similarities emerge among per-node weights. Let us pick two applications from our sample and compare the

respective worker weights, as obtained by hill climbing, node by node. If we multiply the weights of one application by some scalar coefficient such that its aggregate worker weight becomes the same as the other application, the per-node weight variance decreases. The same occurs by following the same procedure for the non-worker nodes. In fact, if we cluster worker nodes and non-worker nodes separately and perform the above scaling on their (clustered) weight distributions, the average per-node coefficient of variation decreases by 1/3.

These key observations enable us to build a practical best-effort solution to bandwidth-aware page placement in asymmetric NUMA systems, which we describe next.

4.3 BWAP: Bandwidth-Aware page Placement

Given a NUMA system and a parallel application running on a set of *worker nodes*, the goal of BWAP is to devise and enforce an efficient interleaving of the application's pages across the NUMA nodes. Since application threads access different memory nodes through potentially diverse bandwidths, BWAP assigns different *weights* to different nodes. A node's weight denotes the fraction of pages mapped to the node.

Figure 4.3 provides an overview of BWAP. BWAP pipelines two key components, the *Canonical tuner* and the *DWP tuner*. The first one is agnostic of the target application and runs offline. The second one is an online component that, at run-time, departs from the first component's output to reach an improved application-specific page placement.

The inner workings of each component can be summarized as follows. The Canonical tuner models our knowledge of the bandwidth of the NUMA topology. Since the effective per-node bandwidth is sensitive to the demand that applications pose on the system, the Canonical tuner assumes an extremely bandwidth-intensive application as its reference. Its output is a set of *canonical weight distributions*, which optimize the memory throughput of the reference application. If one considers different scenarios where the application runs on different sets of worker nodes, the corresponding optimal weight distributions might also differ. Hence, the Canonical tuner considers different combinations of worker nodes as input and accordingly computes the corresponding canonical weights.

As discussed in Section 4.2, the optimal weight distribution varies sub-

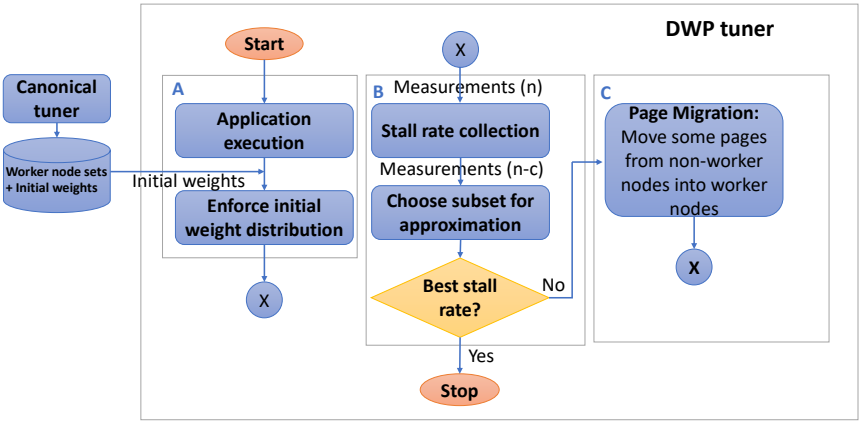


Figure 4.3: Control flow of BWAP.

stantially across distinct workloads. Therefore, the canonical weight distributions, as produced by the Canonical tuner, may not be suited to other applications than the idealized bandwidth-intensive application. Hence, for a given worker node set, the canonical weight distribution is used as a hint about the relative weight distributions to be employed for the worker set and the non-worker sets of the target application. Then, by leveraging Observation 3 (Section 4.2), the DWP tuner converts the canonical weight distribution to one that is optimized for the target application. This is done by finding an appropriate value for a *data-to-worker proximity factor* (*DWP*), which determines how many pages will be assigned to the set of worker nodes (retaining the canonical weight relations), while the remainder will be shared within the non-worker nodes (also according to the canonical distribution). The DWP tuner achieves this through an incremental page migration mechanism that searches for a good value of *DWP* for the target application. This stage is done on-the-fly, during the first seconds of the actual execution of the application.

The following sections detail each component of BWAP.

Calculating canonical weights

First, the Canonical tuner models the underlying machine and an idealized canonical application running on it. Using the model, the Canonical tuner computes the canonical weights.

System model

The Canonical tuner makes a set of simplifying assumptions on the underlying machine and the target applications. We show in Section 4.4 that this model suffices for BWAP to be an effective best-effort approximation, even when the assumptions we introduce next are not entirely met in reality. We assume a cache-coherent NUMA machine comprising a set of N nodes, $\mathcal{N} = \{n_1, n_2, ..., n_N\}$, where the set of nodes as a whole is entirely managed by a single instance of an operating system. Each node contains one or more multi-core CPUs, which globally provide C hardware threads. For simplicity, we assume that every node's local computing and memory resources (like CPU frequencies, number of cores, or local memory bandwidth) are identical. Furthermore, each node includes one or more memory controllers providing access to the local memory of that node. We abstract the memory controllers that are local to a given node as one single-channel memory controller, whose bandwidth is the aggregate bandwidth of each channel/memory controller in the real topology.

Threads may read and write pages that reside on the local node's memory, and on any remote node's memory. In the latter case, the read/write request is sent through the interconnect, which provides full connectivity among all nodes.

The interconnect topology is asymmetric, i.e., a thread running on a given node will observe different bandwidths and latencies, depending on the memory node it is accessing. The most obvious difference in bandwidth and latency is between local and remote accesses. Among remote accesses, different bandwidths and latencies may be observed, since distinct interconnect links may have distinct bandwidths (possibly distinct bandwidths for each communication direction), and paths between nodes can differ in number of hops (some nodes are directly connected, while others communicate through multi-hop paths).

On top of the NUMA system characterized so far, our model assumes a simplified parallel application, which we refer to as *canonical application*,

that uses t threads, where $t \leq C \times N$. Threads are placed at a subset of nodes, $\mathcal{W} \subseteq \mathcal{N}$, which we call *worker nodes*. For simplicity, we assume that t is a multiple of the number of worker nodes, and the threads of the canonical application are evenly distributed across the worker nodes. We assume that parameters t and $\mathcal{W}$ have been previously tuned by some existing tool(s) for optimal parallelism tuning (tuning of t) and thread placement (tuning of $\mathcal{W}$), e.g., [11, 112]. In addition, the above parameters do not change while the application runs, and no other processes are co-located in the canonical application's worker nodes.

Further, we consider that the work performed and the memory access patterns are similar among all the threads of the application. Within the application's address space, we distinguish between thread-private pages and shared pages. We assume that the access volume to thread-private pages is negligible when compared to the available memory bandwidth.

In contrast, we assume that the canonical application places an extreme memory demand on the shared pages. Moreover, the workload of the canonical application is bandwidth-intensive, such that the memory access throughput that it attains is the dominant factor to its overall performance, rather than memory access latency (and other overheads unrelated to memory). Hence, hereafter we focus on memory throughput to shared pages, and omit latency from our equations. Furthermore, we assume that the canonical application accesses shared data predominantly in read-only mode; i.e., write accesses to shared pages are so rare that they have no relevant impact on performance. Finally, we consider that the canonical application accesses all shared pages with the same probability.

Shared pages are interleaved across all memory nodes in a weighted fashion, where some nodes may receive more pages than others. No matter which page interleaving is chosen, we assume the shared space fits the available physical memory. Pages are interleaved according to a weight distribution, $\mathcal{D} = \{w_1, w_2, .., w_N\}$, where w_i denotes the fraction of shared pages that node i will hold (such that $\sum w_i = 1$). Therefore, to maximize the performance of the canonical application, we need to find the optimal weight distribution that maximizes the overall throughput to shared data.

Finding the optimal weight distribution

Now we discuss how to find the optimal weight distribution for the canonical application. We introduce the function $bw(n_{src} \rightarrow n_{dst})$, which denotes the

bandwidth that a given thread located at worker node n_{dst} can use when reading from a node n_{src}. For presentation simplicity, we start by assuming that function $bw(n_{src} \rightarrow n_{dst})$ is well known for all node pairs in the system and does not change for different weight distributions. We discuss how to lift these assumptions in the next section.

Single-worker scenario. The simplest scenario is where all threads run on a single worker node, n_w. BWAP adapts the approach that Yu et al. proposed in the context of data placement in SMP systems with heterogeneous DRAM-NVM memory hierarchies [40]. We start by considering that the threads in n_w need to read a total of S bytes of shared data before completing their work. Since the canonical application's performance is dominated by memory throughput, its execution time is determined by the time that the threads in n_w take to transfer S bytes from memory. Since memory is interleaved among the memory nodes of the system, threads will need to read from pages held at distinct nodes. Furthermore, since the canonical application accesses any page with a uniform probability, the portion that is read from each node i will be proportional to the weight of i; more precisely, $S \times w_i$ bytes.

Moreover, we consider that the data sets that n_w reads from each node in the system are transferred in parallel to n_w. As Yu et al. have shown [40], this approximation is relatively accurate for memory-intensive applications in systems where the number of accessing threads in a node is substantially higher than the number of source memory nodes.

Putting it all together, we can then determine the execution time of the threads in n_w as the time to complete the longest (parallel) transfer from every memory node:

$$T = \max_{i \in \mathcal{N}} \frac{S \times w_i}{bw(n_i \rightarrow n_w)} \tag{4.1}$$

If pages were uniformly interleaved (i.e., equal weights for all pages), execution time would be determined by the time to transfer from the node with the lowest bandwidth (to n_w). Hence, to minimize execution time, one can reduce the pages placed in that node (by decreasing the corresponding weight) until it no longer incurs the longest transfer time. After that, another node becomes the one that contributes with the longest transfer time and its weight should also be reduced; and so forth. It is easy to show that the resulting optimal solution consists of setting the weight of each node n_i as follows:

$$w_i = \frac{bw(n_i \rightarrow n_w)}{\sum_{i \in \mathcal{N}} bw(n_i \rightarrow n_w)} \tag{4.2}$$

Multi-worker scenario. We now generalize the previous solution to scenarios where the canonical application has a higher parallelism level, thus spans its threads across two or more worker nodes. In this scenario, the execution time of the application is given by the time that (the threads running at) the slowest worker node takes to complete, as follows:

$$T = \max_{n_w \in \mathcal{W}} (\max_{i \in \mathcal{N}} \frac{S \times w_i}{bw(n_i \rightarrow n_w)}) \tag{4.3}$$

Like in the single-worker scenario, we can minimize the execution time by adjusting the weight distribution. However, the multi-worker scenario introduces a subtle challenge: the bandwidth that two distinct worker nodes, n_A and n_B, may use when reading from a target node n_i can be different. In fact, changing the weight of n_i to greedily optimize n_A's performance can degrade n_B's performance, and vice-versa. Therefore, the optimization of the weight distribution needs to take all the transfers by all the worker nodes into account.

We achieve this by defining *minimum bandwidth* as the bandwidth of the weakest path among the paths interconnecting a given target node to the worker nodes as $minbw(n) = \min_{n_w \in \mathcal{W}} bw(n_i \rightarrow n_w)$, and by transforming Equation 4.3 to employ this notion:

$$T = \max_{i \in \mathcal{N}} \frac{S \times w_i}{minbw(n_i)} \tag{4.4}$$

Finally, we can apply a similar optimization strategy as we do in the single-worker scenario, but this time by considering the minimum bandwidth values. Consequently, the optimal solution in the multi-worker case consists of making each node's weight proportional to its minimum bandwidth:

$$w_i = \frac{minbw(n_i)}{\sum_{i \in \mathcal{N}} minbw(n_i)} \tag{4.5}$$

Estimating bandwidth

The solution built so far assumes that the bandwidth between two nodes ($bw(n_{src} \rightarrow n_{dst})$) is well known and does not change for different weight distributions. We now question these assumptions and propose a practical solution that does not depend on them.

It is important to understand what main factors contribute to the bandwidth that threads in a NUMA machine may effectively use. The value of $bw(n_{src} \rightarrow n_{dst})$ is limited by the nominal bandwidth of the path from n_{src} to n_{dst} (if $n_{src} \neq n_{dst}$). However, the effective bandwidth that threads in n_{dst} get when reading from n_{src} is also determined by the access demand that the application places on the system, in different complex ways.

First, it is known that the actual bandwidth of a single memory controller is influenced non-linearly by the actual access demand to that controller [143]. To complicate matters, the effective bandwidths, as perceived from worker nodes, are also strongly affected by cross-thread and cross-node interference [144]. The access demand on the memory controller at node n_{src} includes concurrent accesses from multiple threads running in the same worker (either n_{src} or a remote node); further, if the canonical application spans multiple worker nodes, the memory demand on n_{src} combines contending accesses by threads residing on distinct nodes (besides local threads at n_{src}). Finally, bandwidth is also affected by interconnect congestion. This may happen when multiple co-located threads access the same remote memory node (through the same link), or when threads from different worker nodes issue memory requests whose result is delivered through interconnect paths that share one or more links. Therefore, it is clear that our initial assumptions on $bw(n_{src} \rightarrow n_{dst})$ are questionable. First, we relied on an exact knowledge of bandwidth in a NUMA machine. Second, when searching for the optimal weight distribution, we assumed that memory throughput was immutable during that search.

BWAP follows a pragmatic approach to approximate the $bw(n_{src} \rightarrow n_{dst})$ function. More precisely, for a fixed set of worker nodes, we start by deploying a memory-intensive benchmark (representing the canonical application) and uniformly interleaving the benchmark's pages across all nodes in the machine. We use a simple benchmark that spawns as many threads as the available hardware threads on the worker nodes, and each thread performs a random traversal of a shared array. At the same time, we rely on hardware performance counters to monitor per-node memory throughput.

The profiled throughputs between each pair of nodes, n_{src} and n_{dst}, are used as the values of $bw(n_{src} \rightarrow n_{dst})$. This approach neglects the differences in access demand that occur when page placement changes from the profile-time uniform interleaving scenario to the final weighted interleaving scenario. However, our results (in Section 4.4) confirm that, BWAP is still able to devise efficient weight distributions.

On the practical side, at installation time on a given machine, the Canonical tuner needs to run the above profiling procedure for the relevant combinations of worker node sets (with different sizes). The set of explored worker node sets does not need to be exhaustive: i) a large number of worker node sets can be filtered out since they are unlikely to be used by a rational user (e.g., in a dual-socket machine with 2+2 nodes, a 2-worker set comprising nodes at each socket, thus interconnected with a low bandwidth); ii) many worker node sets are symmetrical, hence only one needs to be configured (e.g., in a dual-socket machine with 2+2 nodes with symmetric links between sockets, the optimal weight distribution for the worker set comprising two nodes on one socket is symmetrical to the set comprising the nodes on the other socket).

On-line page placement tuning

The DWP tuner takes action when an application is launched. The tuner's API includes a main function `BWAP-init`, which should be called by the target application once it has allocated its initial shared structures. Note that DWP tuner targets applications, which, after an initial stage, enter an execution stage with stable memory access behavior (identically to systems such as Carrefour [12] and Asymsched [11]). The main argument of `BWAP-init` points to the set of worker nodes on which the application is running.

The goal of the DWP tuner is to tune the weight distribution by searching for an appropriate application-specific DWP. We recall that DWP determines the balance between pages mapped to the set of worker and non-worker nodes, while preserving the relative weight relations within the sets of worker and non worker nodes. The canonical weight distribution corresponds to $DWP = 0$, while $DWP = 1$ corresponds to the extreme where all pages are mapped to the worker node set. This allows BWAP to be used with both bandwidth-sensitive (low DWP) and latency-sensitive workloads (high DWP).

Tuning data-to-worker proximity

Initially, the DWP tuner obtains a pre-computed canonical weight distribution for the worker node set. The application's shared pages are initially placed based on this weight distribution ($DWP = 0$).

The BWAP library adapts *DWP* on-the-fly during the first seconds after the application calls `BWAP-init`, based on hill climbing. Departing from the $DWP = 0$, we periodically monitor average resource stall rates (stalled cycles per second), by reading hardware performance counters via a portable library [145]. It is well known that stalled cycles are strongly correlated to execution time [142]. At each period, we collect n measurements over an interval of t seconds. We then sort and discard the first and the last c measurements to filter outliers. At each iteration, we compare the current average stall rate with the previous one, and accordingly vary *DWP* by a constant step, x. If the stall rate decreased, it is likely that increasing *DWP* improved the overall performance. Hence, we continue increasing *DWP*. Otherwise, it is likely that we found a (local) optimum and we stop the hill-climbing.

At each iteration where we decide to increase *DWP*, applying the new value is ensured by incrementally migrating pages from non-worker nodes to worker nodes, where the number of pages that is removed from/added to a given node is proportional to that node's canonical weight. This ensures that the relative canonical weights within the worker and non-worker node set are preserved as *DWP* changes, as intended.

Initial placement and incremental migration

At each iteration, the DWP tuner needs to place pages according to a weighted interleaving strategy. However, no direct support for *weighted* interleaving exists in mainstream OSs. Hence, we have to build such support for weighted-interleaved page placement. We achieve this by two alternative means: at kernel and user levels. At the kernel level, we implemented a new policy to support weighted interleaved memory allocation, exposed by a new system call. We also added the weighted interleave option to *numactl* tool and *libnuma* library to avoid the burden of application-level changes.

Our user-level alternative has the advantage of portability (avoiding the need for patching the underlying kernel), yet at the cost of a less accurate interleaving. Algorithm 3 summarizes our user-level page placement algorithm. We start by determining the currently allocated page ranges that are likely to hold shared data. This includes the *.data* and BSS segments, as well as dynamic memory mappings. We divide each address range into contiguous sub-ranges and, for each sub-range, we call *mbind* with the uniform interleaving option over a different set of nodes (line 8 in Alg. 3).

Algorithm 3: User-level weighted interleaving approximation used by the DWP tuner

```
   Input: segment // Struct with start address and length
   Input: nodes // Set of NUMA nodes and respective weights
 1 begin
 2     address ← segment.startAddress();
 3     weight_prev ← 0;
 4     while nodes ≠ ∅ do
 5         node ← getNodeWithMinWeight(nodes);
 6         weight ← node.weight − weight_prev;
 7         size ← |nodes| ∗ weight ∗ segment.length();
 8         interleave_uniform(address, size, nodes);
 9         nodes ← nodes − {node};
10         address ← address + size;
11         weight_prev ← node.weight;
12     end
13 end
```

More precisely, the first sub-range's pages are interleaved over all nodes, the second sub-range's pages are interleaved over all nodes except the one with the lowest weight, and so forth. The key insight is that, by setting the size of each sub-range (line 7 in Alg. 3), we can ensure that the overall per-node page ratios will be proportional to the desired weights.

This solution is not as accurate as the kernel-level alternative, since it does not enforce that all the sub-ranges reflect the weighted interleaving. Still, it ensures a best-effort shuffling of pages, while keeping the number of *mbind* calls low. By using the *MPOL_MF_MOVE* and *MPOL_MF_STRICT* flags of *mbind*, this approach also works correctly (without kernel modifications) when weight distributions are changed dynamically by the DWP tuner. It is easy to show that, as *DWP* increases in each step, Algorithm 3 will call *mbind* on sub-ranges that were previously mapped according to the same or a wider interleaving. In this case, *mbind* seamlessly performs the necessary page migrations. The reverse migration/operation is currently unsupported by *mbind*.

Co-scheduled variant

The mechanism described so far assumed a stand-alone application that runs on a subset of nodes and also has the remaining nodes idle to place its pages. However, as mentioned in Section 2.3, many NUMA systems will consolidate workloads in a single physical machine in order to minimize idle hardware resources. Workload consolidation is today an active research topic, both in academia and industry (e.g., with Intel's recent introduction of Resource Director Technology [28] into its high-end processors). Recall from Section 2.3 that a common formulation of the problem considers one or more *latency-critical* applications (LCAs) that are supposed to perform as well as if they were accessing isolated resources (e.g. memory); and one or more *best-effort* applications (BEAs) that benefit if provided with some resources that are originally assigned to the former workloads [10, 15, 16, 92]. Recent proposals to this problem [10, 14–16, 57, 92] focus only on single-socket scenarios. In BWAP, we improve on this by enabling the allocation of the bandwidth of the full set of NUMA nodes across two or more applications running in disjoint worker nodes.

We consider a simpler allocation problem (than the one presented in Section 2.3), where every application is seen as best-effort (i.e., no SLOs), but some BEAs are high-priority and others are low-priority. Specifically, we can consider one high-priority BEA, A, that has a low memory intensity; and a low-priority BEA, B, that is memory-intensive. It is desirable that B places some of its pages on the nodes where A runs (i.e., B's non-worker nodes), so B can benefit from the spare bandwidth of those nodes. However, such memory consolidation strategy should not degrade A's memory performance. To support this scenario, the DWP tuner supports a co-scheduling variant, where the iterative search comprises two stages, both coordinated by an external process that monitors the stall rates of both applications. The rationale of this 2-stage approach is that, below a given DWP of B, the demand that A will pose on the local nodes of A will start having a noticeable degradation of A's performance. Hence, the search should only consider DWP values that are above that lower bound.

The first stage determines an approximation of such bound. The monitoring process measures the stall rate of A, increasing the DWP of B as long as the stall rate of A keeps decreasing. When A's stall rate stabilizes, the search has found a local maximum of A's performance, which is probably a good approximation of the lower bound on the DWP of B. At that point

on, the second stage starts and proceeds as described in Section 4.3-2 (i.e., now guided by the stall rate of B).

As a final remark, we note that there are two aspects that the DWP tuner does not currently handle automatically. First, in the co-scheduled variant, we assume that some external tool/hint [12] has classified each workload as memory-intensive or not. Second, the programmer is expected to call BWAP-init when the program is about to enter its stable phase. The first limitation can be addressed by using the number of memory accesses per instruction (MAPI) to classify workloads as either memory-intensive or not (like in Carrefour [12]). As for the second limitation, one may consider looking at the periodic variation of the MAPI metric and only trigger the DWP tuner when such variation is below a given threshold.

4.4 Evaluation

Our evaluation answers two key questions: *1. What performance advantage does BWAP bring to memory-intensive applications on NUMA systems compared to state-of-the-art page placement policies like Carrefour [12] or Asymsched [11], which rely on uniform interleave to place shared pages? 2. Considering each main component of BWAP separately, how effective is it and what overheads does it introduce?*

We consider several state-of-the-art page placement policies, i.e., the Linux's default policy (*first-touch*), uniform interleaving across workers (*uniform-workers*), uniform interleaving across all nodes (*uniform-all*), and *autonuma* [117].

Among the different policies evaluated, *uniform-workers* is especially relevant since it is the core strategy that state-of-the-art proposals adopt when placing pages across memory nodes of a NUMA system. Among others, this includes proposals like Carrefour [12], Asymsched [11], and the bandwidth-aware policies proposed by Baek et al.[40]. Since other recent bandwidth-aware page placement proposals do not support NUMA (e.g. [38, 39]) they are not represented here.

Although Carrefour and Asymsched resort to *uniform-workers*, they complement it with two main optimizations: namely the detection and co-location of private pages, and the replication of read-only pages. We do not evaluate these features since they require kernel modifications and no patch was available for our operating system versions. Still, we note that such op-

Table 4.1: Memory access characterization of the evaluated benchmarks, as obtained by NumaMMA [146] tool on machine B running each benchmark on a full worker node. The trends of each benchmark do not change significantly for higher number of workers (results omitted for space limitations).

Benchmark	BW Requirements		Memory Access Pattern	
	Reads (MB/s)	Writes (MB/s)	Private Accesses (%	Shared Accesses (%)
Ocean_cp (OC) [34]	17576	6492	79.3%	20.7%
Ocean_ncp (ON) [34]	16053	5578	86.7%	13.3%
SP.B [1]	11962	5352	19.9%	80.1%
Streamcluster (SC) [33]	10055	70	0.2%	99.8%
FT.C [1]	5585	4715	95.0%	5.0%

timizations are orthogonal to our paper, since they can directly complement the mechanisms proposed in BWAP.

Besides these alternatives, we evaluate complete BWAP and an incomplete variant, denoted *BWAP-uniform*, which disables the Canonical tuner. This variant departs from *uniform-all* as its initial weight distribution and only runs the DWP tuner.

For space limitations, the weighted page interleaving in BWAP is enforced with the portable, user-level option. By enabling the kernel-level variant, we observed only marginal gains (at most 3%) and reach the same main conclusions. The parameters of the DWP tuner of BWAP (Section 4.3) are set as follows: $n = 20$, $c = 5$, $t = 0.2$, and $x = 10\%$. We chose these values by optimizing the parameters for a particular setting (benchmarks Ocean_* on machine A), then used those values for every other benchmark/machine/experiment.

For our evaluation, we have selected memory-intensive benchmarks from NAS [1], PARSEC [33] and SPLASH [34] suites. We intentionally omit benchmarks with low memory intensity, since page placement has marginal impact on their performance. Although BWAP's design assumes read-only and shared-only pages, we evaluate how it performs as a *best-effort* approach with workloads that do not fit those assumptions. Therefore, our selection of benchmarks promotes diversity with regard to the ratios of read vs. write accesses, and thread-private vs. shared accesses. The memory access characteristics of each benchmark is shown in Table 4.1, which confirms that

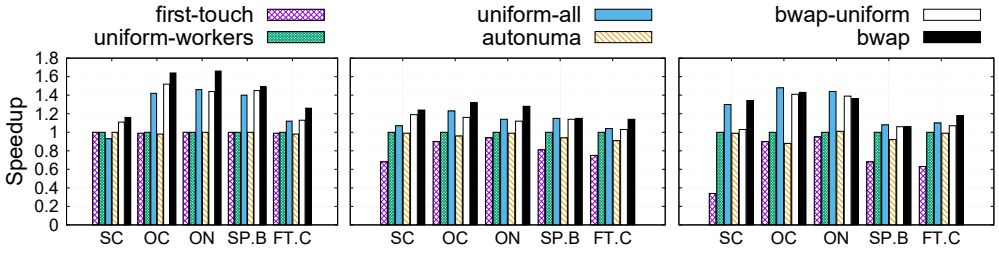
most benchmarks are far from the assumptions underlying BWAP's design. Specifically, three of the benchmarks: OC, ON and FT.C – have more than 79% of thread-private memory accesses. Benchmarks also differ in their scalability, as Table 4.4 summarizes. As for the datasets, we used the largest available datasets, i.e., the native inputs for PARSEC and SPLASH and the CLASS B and C datasets for NAS. Each of these datasets fits in the memory of each node. We use execution time, averaged over 5 runs, as the performance metric.

All the benchmarks are multithreaded applications with a malleable thread pool. For thread placement, we used the usual rule of thumb that is adopted, for instance, by AsymSched [11]: threads are grouped together on the subset of worker nodes with the highest aggregate inter-worker bandwidth. To minimize scheduling-related overheads (e.g., core over-subscription, simultaneous multithreading, thread migration), we pin each thread to a distinct core. We leave the interaction of page placement and OS-guided thread scheduling as future work. The page size used in our experiments is the Linux default page size (4KB). Integrating BWAP with large pages in NUMA systems [147] is left as future work.

We ran our experiments on 2 NUMA systems of different scales and asymmetry. Machine A is a 4-socket AMD Opteron Processor 6272, with 8 memory nodes, 8 cores per node, 64GB DRAM, running Linux 4.17. Machine A is representative of a high-end NUMA system with a strongly asymmetric interconnect topology. As it is evident in Figure 4.2a, interconnect links exhibit ample disparities in link bandwidth, sometimes affecting even different directions of the same link. Machine B is a 2-socket Intel Xeon CPU E5-2660 v4. In contrast, Machine B is a smaller-scale machine with a simpler topology. It has 4 NUMA nodes (using Cluster-on-Die mode), 7 cores per node, 32GB DRAM, running Linux 4.4. In this case, memory bandwidth still exhibits asymmetries, however less pronounced than in machine A. While the lowest bandwidth in machine A was 5.8× lower than the highest bandwidth (i.e., local memory bandwidth), that amplitude drops to 2.3× in machine B.

Overall performance evaluation

To compare the performance of BWAP to page placement alternatives, we measure the performance of benchmarks when using different placement



(a) One worker node (b) 2 worker nodes (c) 4 worker nodes

Figure 4.4: Speedup vs *uniform workers* (co-scheduling, machine A)

policies. We focus on two representative execution scenarios: *co-scheduled* and *stand-alone*.

Co-scheduled scenario. In this scenario, two benchmarks share the same NUMA machine (for instance, hosted by a Cloud provider), as assumed in Section 13. We assume that one benchmark, A, is not memory-intensive and the other, B, is. Hence, while A will place pages locally for improved latency, B will resort to BWAP to scatter its pages across all nodes (including the non-worker nodes where A resides) in order to optimize B's throughput, while not degrading A's.

Figures 4.4, 4.5b and 4.5a compare the performance of each application B on the co-scheduled scenario. We consider different allocations of worker nodes to application B: 1, 2, and 4 worker node(s) in machine A; 1, and 2 worker node(s) in machine B. In all cases, A runs on the remaining nodes. As for application A, we run Swaptions [33]. In all our co-scheduled experiments, we did not observe relevant changes to the performance of Swaptions when application B placed some of its pages on the nodes of A, so we omit the performance results of this application in this analysis.

Stand-alone scenario. In this scenario, the NUMA machine is entirely available for application A, which can be deployed with any number of threads/workers. Hence, a rational user will run the application with its optimal parallelism level (assumed to be tuned *a priori*). Note that, for each application, its optimal parallelism level may differ depending on the page placement policy. Figures 4.5c and 4.5d show how each application performs when running with the different page placement alternatives in

Worker nodes	Corresponding weights i.e. $w1$,$w2$,.. (%)	Workers variance	Non-workers variance
0	29.2, 17.4, 12.8, 11.4, 8.9, 5.9, 8.7, 5.7	N/A	17.111
0,1	19.3, 19.2, 14.3, 14.3, 8.3, 8.6, 7.7, 8.3	0.005	9.927
0,1,2,3	9.8, 15.9, 16.1, 16.1, 11.4, 10.9, 9.6, 10.2	9.723	0.623
All	14.7, 12.7, 12.8, 14.7, 11.6, 11.1, 10.6, 11.8	2.382	N/A

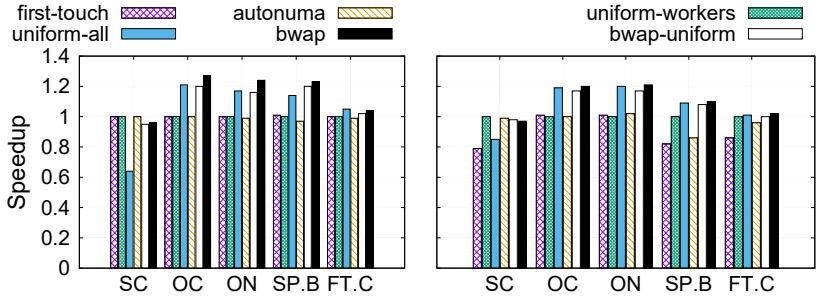
Table 4.2: Canonical weights computed by the Canonical tuner for varying number of workers for machine A. The weights are in an ascending order with respect to node numbers i.e. $w1$ represents weight for node 1, $w2$ represents weight for node 2 and so forth.

this scenario. The performance of the *BWAP-uniform* variant is discussed in Section 4.4.

Analysis of the results. The results for both scenarios confirm that, as expected, the best performing solutions are those that fully exploit the available memory bandwidth by placing shared pages across all nodes (*uniform-all* and BWAP), rather than restricting placement within the boundaries of the worker node set (*uniform-workers* and *autonuma*).

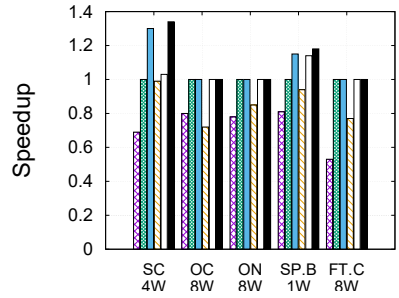
Unsurprisingly, *first-touch* is usually the worst alternative for multi-worker scenarios by substantial margins, since it tends to centralize many shared pages on a single node (where the initializing threads run) as studied before [12, 148, 149]. Our results show that, even in a single worker scenario, binding the entire application's memory locally to the only worker node is suboptimal for memory-intensive applications. Among the solutions that spread pages across all nodes, BWAP achieves the best performance or, with less favourable applications, performs comparably to the best solution (*uniform-all*). More precisely, BWAP is able to outperform both *uniform-workers* and *autonuma* by up to 1.66×, and *uniform-all* by up to 1.50×. As expected, the largest speedups of BWAP are observed on machine A, which has the most asymmetric topology. This confirms our main claim that, for some types of workloads, trivial uniform interleaving policies are not appropriate to exploit the full bandwidth of complex NUMA systems.

One important trend is that the benefits of BWAP over the uniform interleaving alternatives drop when more workers are involved. This is evident in the co-scheduled scenarios (as the number of workers increases). Further, in the stand-alone scenario, this explains why BWAP only achieves

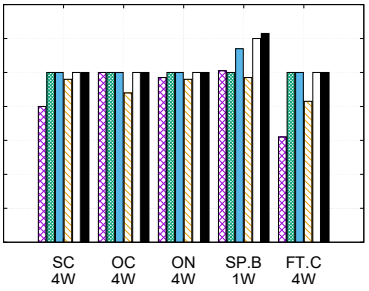


(a) One worker node (co-scheduling, machine B)

(b) Two worker nodes (co-scheduling, machine B)



(c) Optimal num. of workers (stand-alone, machine A)



(d) Optimal num. of workers (stand-alone, machine B)

Figure 4.5: Speedup vs *uniform workers*

relevant gains for those applications whose optimal parallelism level is lower than the total number of nodes. This trend can be explained by two factors. First, as applications use an increasingly larger worker node set, the worker vs. non-worker dichotomy fades away. Thus, the gains from tuning *DWP* decrease. A second, less intuitive factor, is that, as one enlarges the worker node set, the inter-worker canonical weight distributions (as devised by our Canonical tuner) tend to uniformity. This trend is clear in Tables 4.2 and 4.3. The main insight behind this is that, although the interconnect topology is asymmetric from the perspective of each node (regarding the individual connections to the other nodes), the set of outgoing and incoming connections of all nodes share similar profiles: for instance, in machine A, every node is connected to 1 *close* node (i.e. very high bandwidth), 2 *very far* nodes (i.e. very low bandwidth), etc. In a configuration with few worker

Worker nodes	Corresponding weights i.e. $w1$,$w2$,...(%)	Workers variance	Non-workers variance
0	33.9,24.6,21.2,20.3	N/A	0.001
0,1	29.6,28.6,22.0,19.8	0	0
All	25.0,25.0,25.0,25.0	0	N/A

Table 4.3: Canonical weights computed by the Canonical tuner for varying number of workers for machine B. The weights are in an ascending order with respect to node numbers i.e. $w1$ represents weight for node 1, $w2$ represents weight for node 2 and so forth.

nodes, it may happen that a pair of *very far* nodes is included in the worker set, but not all pairs, which will introduce asymmetry and induce diverse weights. In contrast, as the worker node set grows, then eventually all *very far* node pairs become part of that set, thereby resulting in a increasingly uniform weight distributions.

Let us now focus on the subset of benchmarks with non-negligible thread-private memory bandwidth demand, namely OC, ON and FT.C. For such benchmarks, consider the set of thread-private pages that belong to the threads running in a given worker node. In theory, the optimal placement of such pages should consider that worker node only (and regard every other node as non-worker). However, both components of BWAP are, by design, based on the simplifying assumption that every page is accessed from *every* worker node. This approximation allows BWAP to decide the placement of every page similarly, independently of each page's actual worker node set (i.e., the full worker node set in the case of shared pages; a specific worker node in the case of thread-private pages).

Despite this approximation, BWAP is still able to obtain important performance gains with those benchmarks whose ratios of thread-private accesses are at odds with the above assumption (OC, ON and FT.C). The performance benefits obtained with these benchmarks follow the same trend as discussed above: the performance advantage of BWAP over the remaining alternatives is especially evident when the set of worker nodes is smaller; it decays as we enlarge the worker set, becoming comparable to the best-performing alternative when worker nodes span across all nodes in the machine.

A more careful analysis allows us to shed some light on these results. We start by observing that the memory demand that these benchmarks place

on thread-private pages is sufficiently high to saturate the local memory bandwidth. We support this observation by the fact that, when we place thread-private pages exclusively locally to the thread that accesses each page (with the *first-touch* policy), their performance degrades relatively to the opposite extreme of *uniform-all*. Therefore, the thread-private access demand of these benchmarks calls for page placement strategies that interleave pages across multiple nodes to maximize the available thread-private bandwidth.

Based on this observation, we note that, among the evaluated strategies that resort to page interleaving to meet the above requirement, BWAP's *best-effort* approach is the most effective one in scenarios where the number of worker nodes is small – either one or two in our experiments. Specifically, when there is only a single worker node, BWAP is trivially accurate for thread-private pages. However, when there are two worker nodes, then BWAP will wrongly decide to place some thread-private pages in the nearest node instead of the local node (unlike to an optimal placement that takes thread-private pages into account). Still, our results suggest that the impact of such an inherent inaccuracy is modest. This can be explained by the fact that the bandwidth ratio between the local node and the nearest remote node is much lower (1.7× in machine A, 1.8× in machine B) than the bandwidth ratio between the local node and the farthest nodes (5.1× in machine A, 2.3× in machine B). Consequently, despite its inaccurate modelling of thread-private accesses, BWAP still outperforms alternative approaches whose placement decisions lead to even larger inaccuracies: those that rely on uniform interleaving do not take bandwidth heterogeneity into account at all (thus it places high number of pages in the farthest nodes); while those that simply place thread-private pages locally fail to exploit the additional bandwidth of the non-worker nodes. Overall, these results suggest that BWAP's approximated approach is accurate enough to be advantageous with a wide set of memory-intensive applications.

Detailed analysis of BWAP's components

We study the gains and overheads of BWAP components.

Canonical tuner. Looking at Figures 4.4 and 4.5, we can compare the performance of the full BWAP with *BWAP-uniform*. The difference between these alternatives denote the effective contribution of Canonical tuner. Our results show that, for most cases where BWAP outperforms *uniform-*

workers and *uniform-all*, the larger speedup slice is due to the Canonical tuner (which is evident by observing how the results of *BWAP-uniform* and BWAP differ). Concretely, enabling the Canonical tuner attains speedups of up to 1.32× relatively to the uniform variant of BWAP. Further, the highest relative gains happen in machine A, which can be explained by its highest asymmetry. In contrast, the simpler and less asymmetric topology of machine B makes BWAP essentially perform similarly to *BWAP-uniform*. This highlights that, in machines with pronounced bandwidth asymmetries, simple approaches based on uniform interleaving or 2-level weighted interleaving (like *BWAP-uniform*) are substantially suboptimal; however, in more symmetrical machines, simpler solutions may be an acceptable choice.

DWP tuner. Continuing the previous analysis, we can compare *uniform-all* with *BWAP-uniform* to infer the actual benefits that the DWP tuner contributes (i.e., departing from *uniform-all* rather than from the canonical distribution). Figures 4.4 and 4.5 show that, while *BWAP-uniform* outperforms *uniform-all* in many scenarios, that is not always the case. This can be explained by taking into account two considerations. First, for some scenarios, the optimal *DWP* is 0. In this case, *BWAP-uniform* produces the same interleaving as *uniform-all*, but with the overhead of the online iterative search. Table 4.4 details the optimal *DWP* for each application in each co-scheduled scenario. This table yields a high correlation between the null *DWP* cases and the cases where *BWAP-uniform* does not outperform *uniform-all*. On the positive side, Table 4.4 also shows many cases where adapting *DWP* allowed *BWAP-uniform* to choose intermediate values that clearly outperform *uniform-all*. This results in performance gains of up to 1.49× relatively to *uniform-all* (which are further increased if we consider the combination of both BWAP components).

Overhead and accuracy of DWP tuner. Finally, we study the accuracy and overheads of the DWP tuner. To evaluate both aspects, we have manually deployed each application (at a given machine and scenario) with different *static* values of *DWP* and measure the corresponding execution times and average stall rates. This allows us to understand, what the optimal *DWP* is, what is the corresponding (maximum) performance, and the stall rate curve effectively guides us towards finding the optimal *DWP*. By comparing with the value that our DWP tuner chooses and the resulting

	Machine A			Machine B	
Application	1 Worker	2 Workers	4 Workers	1 Worker	2 Workers
SC	48.00%	0%	23.80%	100%	100%
OC	14.10%	0%	0%	0%	0%
ON	14.10%	16%	0%	0%	0%
SP.B	0%	0%	0%	15.20%	22.20%
FT.C	0%	16.30%	0%	30.30%	0%

Table 4.4: Ideal number of worker nodes and DWP values computed via BWAP iterative search (Co-scheduled scenario).

execution time, we can assess how close to optimal our search gets and at which overhead.

Our complete experiments with the entire selection of applications confirmed that stall rate is effectively correlated to execution time and its variation with *DWP* is essentially convex. Furthermore, the DWP tuner was able to successfully find the optimal *DWP* by a maximum error margin of 1 iterative step for all cases we evaluated. Figure 4.6 illustrates this with the Streamcluster application on machine A.

Regarding execution overhead, in the experiments discussed in Section 4.4, we measured a maximum overhead of 4% over all the applications. We found that the overhead of DWP tuner can be increased by two main factors: i) higher optimal values of *DWP*, since the search starts at the opposite extreme and each iteration costs time and page migrations; and ii) lower execution times, which do not allow to amortize the cost of the search and benefit from resulting optimized page placement.

User-level vs. Kernel-level placement mechanisms. As Section 4.3 discusses, user-level alternative has the advantage of requiring no changes to the kernel, yet at the cost of a less accurate interleaving. However, our results with our selection of applications suggest that this theoretical difference in accuracy of each page placement alternative does not have a relevant impact on the overall performance. More precisely, overall, kernel-level placement is able to outperform user-level placement by up to 3%. Figure 4.7 illustrates this with the Streamcluster application.

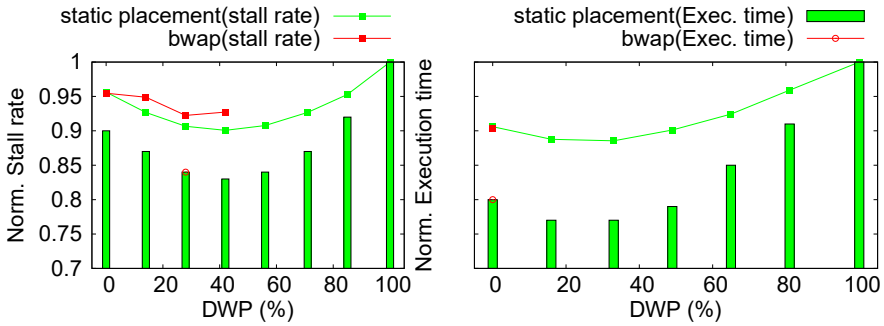


Figure 4.6: Evaluating the *DWP* iterative search using Streamcluster on machine A, with 1 (left) and 2 worker nodes (right).

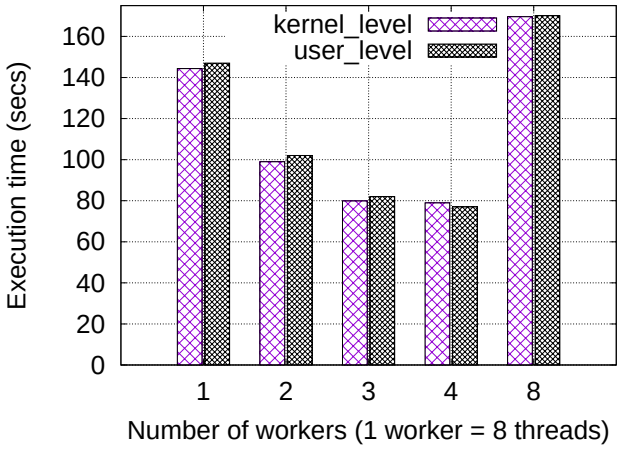


Figure 4.7: Comparing the performance of using the kernel-level vs. user-level weight page placement mechanisms in BWAP-lib with Streamcluster on machine A with different number of worker nodes.

4.5 Related Work

When deploying an application on a NUMA system, a number of complex questions arise, from how many threads, to where to place threads and pages. As NUMA systems increase their prominence, these problems receive increasing attention from the research community. Optimizing thread and memory placement on NUMA systems has been extensively studied [11, 12,

112, 113, 116, 119, 150–158].

Linux provides extensions[117, 121] to improve data access locality in NUMA systems. However, these extensions do not account for the interconnect asymmetry and do not improve bandwidth for communicating threads. For instance, autonuma[117] implements locality-driven optimization by migrating threads closer to the memory they access, and/or by migrating data to the memory closer to the threads. Linux also provides an option to uniformly interleave part of address space across memory nodes. BWAP complements these strategies by providing a novel bandwidth-aware alternative to the uniform interleaving.

Considering proposals for page placement, we distinguish proposals aiming at minimizing memory latency (e.g., [113, 117]) and proposals aiming at optimizing memory bandwidth (e.g., [11, 12]). Depending on the application, the main bottleneck can be latency or bandwidth, or somewhere between those extremes. In contrast to the proposals that focus on either of extremes, BWAP takes this spectrum into account with its DWP tuner.

Apart from page placement, other approaches have been proposed to improve memory performance in NUMA systems. Carrefour [12] replicates read-only pages accessed from multiple nodes. It also detects private pages and places them close to the corresponding thread. Shoal [148] and Smart Arrays [149] introduce programming abstractions that enable a runtime layer to choose the most appropriate page placement, data layout or replication, while taking into account the underlying memory topology and the runtime behaviour of the application. Shoal and Smart Arrays adopt uniform interleaving as one of their NUMA-aware page placements. BWAP is less intrusive, since the application only needs to be linked with and activate DWP tuner. Still, the design principles behind BWAP's placement policy can also be applied to improve these systems.

Recently, some authors have studied the problem of bandwidth-aware page placement for different heterogeneous memory systems [38–40, 159, 160]. All these works consider a tiered memory architecture consisting of two or more types of memory nodes with heterogeneous performance characteristics (bandwidth, latency, etc). Moreover, all works share the fundamental insight that, performance of bandwidth-intensive applications improves if memory accesses are distributed across all memories, proportionally to the bandwidth of each memory. NUMA systems, as addressed by BWAP, are also characterized by heterogeneous bandwidths, even when every NUMA node relies on the same memory technology (e.g., DRAM).

Hence, BWAP shares the same bandwidth-aware placement principle with the previous works. However, the problem of bandwidth-aware page placement in NUMA systems has crucial differences (as Section 4.1 highlights), with new challenges that render the above-mentioned proposals either inappropriate or strongly suboptimal for NUMA systems. The aforementioned proposals for bandwidth-aware page placement either do not support NUMA systems [38] or use the `uniform-workers` policy to distribute pages across (hybrid memory) NUMA nodes. Hence, BWAP can complement or extend these techniques to support NUMA systems whose nodes comprise heterogeneous hybrid memory hierarchies.

This work is also related to workload consolidation techniques for cloud computing systems and data centres. Recent works have focused on optimizing memory throughput in these scenarios by employing last-level cache and memory bandwidth partitioning [14–16, 57] across co-scheduled applications. While these works focus on single-node systems where two or more applications share the same CPU, as discussed in Section 2.3, BWAP targets multi-node NUMA systems where each node is exclusively allocated to one application at most. Works such as [31, 32] propose effective tools to characterize (either through an analytical model or through an empirical procedure) the NUMA topology. These can be integrated into BWAP to allow BWAP to devise a more accurate canonical distribution.

4.6 Summary

Although new thread placement approaches for asymmetric NUMA systems have recently emerged, today's usual techniques for page placement still rely on the obsolete assumption of a symmetric architecture.

This chapter presented BWAP, a novel approach for asymmetric bandwidth-aware placement of pages in NUMA systems. Our evaluation shows that BWAP improves the gains of state-of-the-art policies by up to 66%, on commodity NUMA machines. The gains of BWAP are especially evident in co-scheduled scenarios and when the application does not scale up to the available hardware parallelism.

As a concluding remark, while far from trivial, the design of BWAP is inherently *best-effort*, since it relies on important simplifying assumptions about the workloads and the underlying system. Therefore, our contributions can be seen as the first step that opens avenues for follow-up re-

search on bandwidth-aware page placement for NUMA systems. BWAP also shows promising benefits in workload consolidation scenarios where two or more applications run in disjoint sockets. However, it does not support dynamic scenarios, where the overall system behaviour may change over time. BWAP also lacks support for QoS. In the next chapter, we present BALM, a QoS-aware memory bandwidth allocation technique for multi-socket cloud servers, to address these shortcomings.

Chapter 5

BALM: QoS-Aware Memory Bandwidth Partitioning for Multi-Socket Cloud Nodes

In the previous chapter, we presented BWAP, which enables bandwidth allocation of the full set of sockets across 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.

In this chapter, we present BALM, to address these shortcomings. BALM is a QoS-aware memory bandwidth allocation technique for multi-socket architectures. The key insight of BALM is to combine commodity bandwidth allocation mechanisms originally designed for single-socket with a novel adaptive cross-socket page migration scheme. Our experimental evaluation with real applications on different dual-socket machines shows that BALM can overcome the efficiency limitations of state-of-the-art. BALM can ensure marginal SLO violation windows while delivering substantial throughput gains to bandwidth-intensive best-effort applications when compared to state-of-the-art alternatives. Some passages in this chapter have been quoted verbatim from [161].

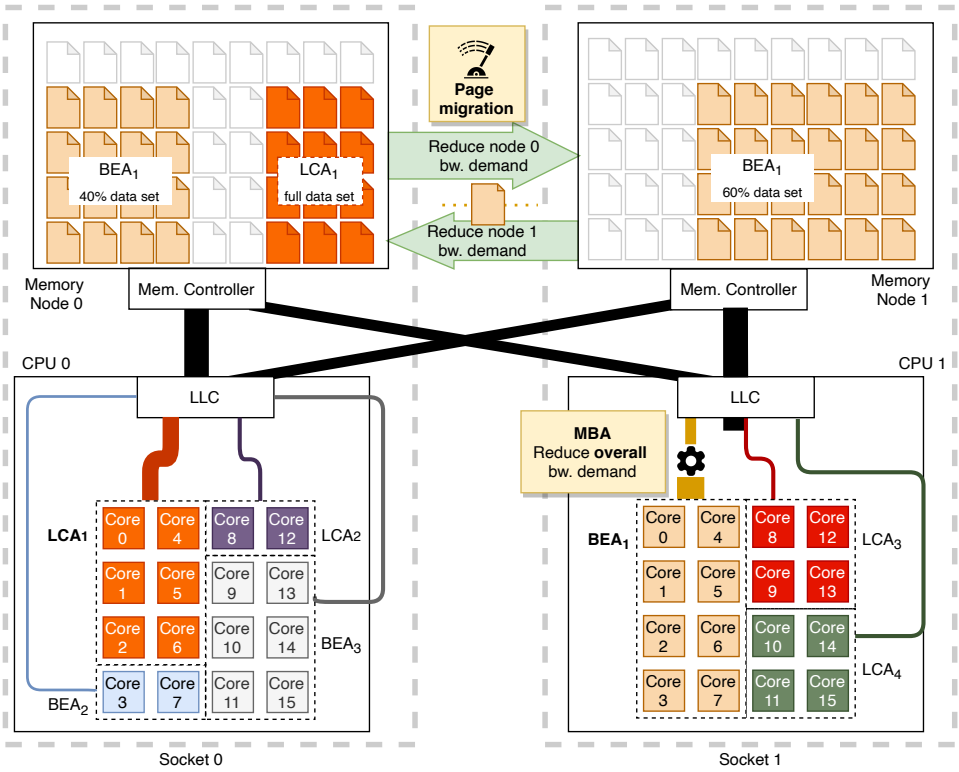


Figure 5.1: QoS-aware bandwidth allocation in a dual-socket across multiple BEAs and LCAs.

5.1 Problem statement

Recent papers [10, 15, 16, 92] 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 work aims at reaching beyond the state-of-the-art by generalizing QoS-aware resource allocation to workload consolidation in multi-socket servers, with a specific emphasis on cross-socket memory bandwidth alloca-

tion. 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 5.1 illustrates.

In a multi-socket system, each socket comprises multiple multi-core CPUs and memory nodes. For presentation simplicity, and without loss of generality, let us 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 memory node, as well as the 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., [11, 116]), 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 in a multi-socket system, 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 [11], ideally with larger fractions of pages in the memory nodes that offer higher bandwidth [140].

We further assume that the LCAs running on a given socket do not saturate the bandwidth of the corresponding local memory node by themselves (i.e., if we exclude the memory usage by BEAs). 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.

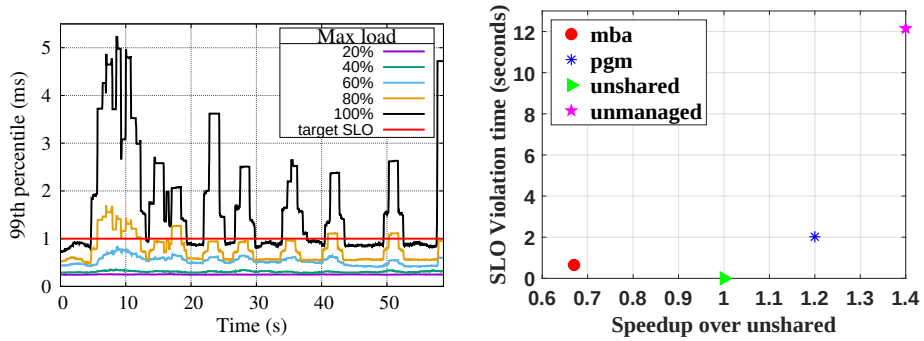


Figure 5.2: Impact of co-locating Memcached (LCA) with Ocean_cp (BEA) on a dual-socket machine (each on a different socket). Left: tail latency of Memcached for different loads; horizontal red line shows the target SLO. Right: Performance of Ocean_cp with different allocation approaches, where Memcached operates at 80% of its max load.

5.2 Performance of MBA and page migration in multi-socket architecture

We start by introducing a simple example that will guide our presentation in the following sections. Let us consider a dual-socket machine, in which we co-locate one LCA, Memcached [35], in socket 0; and one BEA, Ocean_cp [34], in socket 1 (Section 5.5 provides details on the machine and workloads). As expected, the LCA places all its pages locally. In this example, we assume its SLO is that the tail latency (99^{th} percentile), as perceived from the client-side, must be at most 1ms. In contrast, the BEA is bandwidth-intensive and, to attain an optimized throughput, benefits from interleaving its pages across both memory nodes (as already studied in Chapter 4).

Initially, the LCA is running under negligible load and, consequently, meeting its SLO. Therefore, the BEA can safely place a portion of its pages on the remote memory node (where the LCA resides), thus taking advantage of the extra bandwidth. In our example, the BEA initially employs a weighted interleaving approach [140] with 60% of its pages mapped locally, and the remaining ones mapped remotely, which we found to maximize the BEA's throughput (see Section 5.5). Figure 5.1 illustrates these initial page mappings.

Later on, the LCA load increases, as more clients connect to it and thus raise the number of requests per second (RPS). When that occurs, the LCA accordingly increases its (local) memory access demand, requiring a higher memory bandwidth to continue satisfying its SLO. However, the bandwidth demand that the BEA is initially placing on the same memory node (due to its remote pages) interferes with the new bandwidth demand of the LCA. Hence, beyond a given load, the LCA no longer meets its SLO. Figure 5.2 shows the sensitivity of the LCA to memory bandwidth allocation under different loads (RPS) and interference. When the LCA operates at low to medium load (e.g., 20%, 40% and 60% of its maximum load), no SLO violations are observed. However, at higher loads (i.e., when more clients connect to it and thus boost the number of requests per second), the bandwidth of socket 0's memory node saturates and SLO violations become evident and even dominant. The peaks in Figure 5.2 are as a result of co-located BEA having multiple phases with distinct memory intensities.

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.

Next, we provide background on existing mechanisms for memory bandwidth allocation. As we show next, all such mechanisms exhibit important shortcomings when employed to implement QoS-aware memory bandwidth allocation in multi-socket architectures.

MBA and other single-socket mechanisms

Recall from Section 2.3 that thread packing [107] and clock modulation [108] are software mechanisms that have been widely-used for partitioning memory bandwidth in single-socket systems. More recently, Intel released MBA as part of the RDT bundle that ships with Intel Xeon Scalable server processors. MBA supports architecture-level memory bandwidth allocation. MBA provides per-core control over memory bandwidth by injecting a delay value to each outgoing request from level two (L2) cache to the LLC, as depicted in Figure 5.1. The MBA setting of each core can be dynamically

adjusted (e.g., the MBA level can be changed from 100% (i.e., no throttling) to 10% in steps of 10%). The rest of the discussion is focused on MBA, since it is one of the main state-of-the-art mechanisms for memory bandwidth allocation [13]. Still, the main conclusions that we draw next are general to any single-socket-based mechanism.

Although MBA is originally designed to allocate memory bandwidth across applications running on a single socket, nothing prevents it from being used in a multi-socket setting.

In fact, to fix the SLO violation in our example, we might employ MBA to throttle down the memory demand of the BEA, as Figure 5.1 illustrates. For instance, one simple approach is to set the MBA level of the BEA on socket 1 to its lowest value ($MBA = 10$). This approach can fix the SLO violations almost instantaneously, in less than 600 ms. However, this method comes with a relevant cost on the performance of the BEA, since MBA is not only slowing down both its remote accesses (as needed to cure the SLO violation), but also its local memory accesses (by the same MBA delay).

To better quantify this performance penalty, Figure 5.2 (right) shows the performance of the Ocean_cp (BEA). It compares the performance of the BEA when set with the lowest value of MBA (set due to an SLO violation) against two extreme approaches: an *unshared* alternative, where the BEA exclusively maps its pages locally, hence memory bandwidth is not shared; and an *unmanaged* alternative, where no partitioning mechanisms are used. (The *pgm* (page migration) bar is discussed in the next subsection.) As expected, *unmanaged* achieves significantly higher throughput than MBA but fails to safeguard the SLO. On the other hand, *unshared* safeguards the performance of the LCA, but it uses each socket's resources sub-optimally. For example, when the LCA operates at a low load, the BEA can substantially benefit by placing some of its pages on the remote memory node without violating the SLO. This benefit comes at a cost when the LCA operates at a high load (where SLO violations are evident).

Hence, partitioning memory bandwidth is far from trivial in multi-socket scenarios and state-of-the-art solutions that we are aware of (such as MBA) are clearly sub-optimal when used outside the usual single-socket case. Therefore, to make QoS-aware workload consolidation practical in a multi-socket system, an alternative approach is needed to increase the system's utilization to reach close to *unmanaged* and, at the same time, safeguard LCA's QoS with minimal vulnerability windows, as achieved with MBA.

Page migration

In theory, when an SLO violation is detected, migrating pages of the noisy neighbour BEA(s) away from the memory node where the victim LCA runs can be an alternative to MBA (or other single-socket mechanisms). Intuitively, if a subset of some BEA's pages is migrated away from the saturated memory node (towards other memory nodes currently with more spare bandwidth), then a fraction of memory accesses by the BEA (those that target locations in the migrated pages) will no longer translate to memory demand on the former node. Therefore, the SLO violation can be fixed by migrating a selection of pages that contains a large-enough fraction of the BEA's near-future accesses.

We borrow this observation from previous works on page placement for multi-socket systems [11, 12, 140]. A recent proposal, BWAP [140], has shown a *weighted* interleaving approach, where each memory node holds a specific fraction of the application's pages, is typically better than an *uniform* interleaving – provided the weights are adequately tuned, considering different factors related to the computer architecture and the memory access behavior of the workload [140]. As detailed in Chapter 4, BWAP's approach departs from an initial weight configuration and, through an online hill-climbing process, either proportionally expands (i.e., increases the weights of the remote nodes) or contracts (i.e., increases the weights of the local nodes) the weight distribution until it finds an optimum. The hill-climbing is guided by observing how backend-bound stall cycles (obtained by hardware performance counters) evolve after each expansion/contraction.

Although BWAP is not originally designed to solve the QoS-aware bandwidth allocation problem in multi-socket systems, it can be converted into a memory bandwidth allocation mechanism, as Figure 5.1 illustrates. When a BEA is suspected to be causing an SLO violation in another LCA, one only needs to run the above online migration process in the direction (expansion or contraction) that decreases the weight on the local memory node of the LCA.

In contrast to MBA (and other single-socket mechanisms), page migration is able to adjust memory access demand on a *per-memory node granularity*. Hence, upon an SLO violation, in theory there is a weighted page interleaving of the BEA that reduces the access demand on the saturated memory node, as needed to fix the SLO violation, but does not (unnecessarily) reduce the demand on the remaining (unsaturated) memory nodes.

This allows this mechanism to fix SLO violations while providing considerable throughput gains to the BEAs. This is evident when, in Figure 5.2, we compare the throughput that page migration can attain when compared to MBA.

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 [162]. The worst-case scenarios arise when the BEAs have large data sets and/or the amplitude of the SLO violation is high. Hence a large number of pages will need to be migrated. Consequently, fixing the SLO violation in our particular example via page migration takes more than 2 seconds (needed to move around 7 GBytes of pages to fix the SLO in this particular example).

Therefore, page migration's latency is higher than that of MBA by many orders of magnitude. Such latency implies prohibitively long SLO violation windows, which is at odds with the requirement that SLO violations should last only for negligible periods. For this reason, page migration is unsuitable to QoS-aware memory bandwidth allocation, if used as a stand-alone mechanism. Not surprisingly, we are not aware of any proposed solution that relies on page migration to solve QoS-aware memory bandwidth allocation.

5.3 BALM

This section presents BALM, a novel approach to QoS-aware memory bandwidth allocation in 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. 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. Section 5.3 first provides an overview of its novel features. Section 44 then describes the main algorithm that puts all such features together.

Overview

The key insight behind BALM is that, by using MBA and page migration together as a 2-dimensional allocation mechanism, we unveil new opportunities to eliminate SLO violations. To illustrate this claim, Figure 5.3 depicts

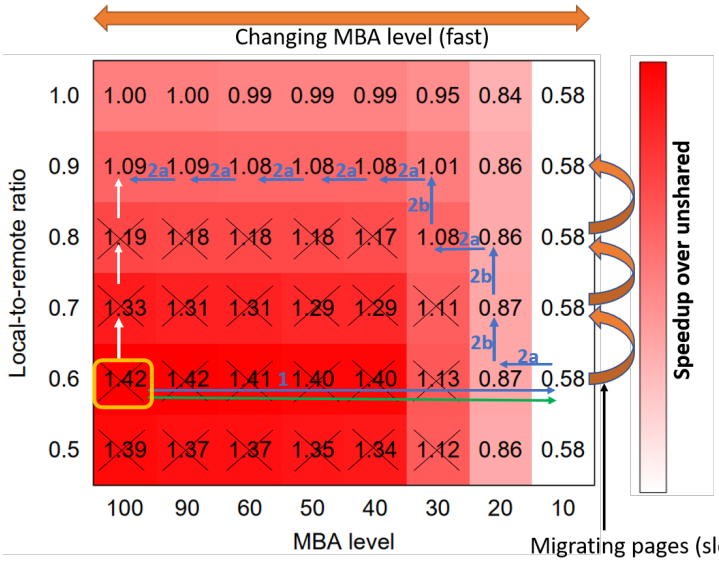


Figure 5.3: Performance of Ocean_cp colocated with Memcached operating at max load on a dual-socket machine. Each cell shows the speedup of Ocean_cp over the *unshared* approach for different configurations. The arrows denote the transitions of different mechanisms when fixing SLO violations: MBA (green), page migration (white), BALM (blue).

the example from Section 5.2 in a 2-dimensional perspective. Each dimension represents the single parameter that tunes each allocation mechanism: with page migration, the ratio between the local and remote interleaving weights (yy axis)[1]; with MBA, the MBA level assigned to the BEA (xx axis). The heat map values at each cell in the matrix denote the BEA's speedup over the *unshared* approach (i.e., the configuration in the top-left extreme).

Recalling the example, the BEA and LCA are deployed at opposite sockets. Initially, the LCA is running under a negligible load. Thus, BALM accordingly chooses the configuration that maximizes BEA's throughput– it places pages in a local-to-remote ratio of 0.6:0.4, with no MBA throttling. Later on, the LCA enters a high load phase. Consequently, some configurations become invalid. These are marked with 'X' in Figure 5.3. Since the

[1]The local-to-remote ratio is the portion of local pages to the portion of remote pages. The figure shows only the portion of local pages.

initial configuration is one of such invalid configurations, an SLO violation occurs. Ideally, it should be fixed by quickly transitioning the BEA from the initial invalid configuration to some configuration that, among the valid alternatives, maximizes the BEA's throughput.

The matrix in Figure 5.3 sheds light on the virtues and limitations of MBA and page migration when used alone to handle SLO violations. Using MBA alone restricts the space of available valid configurations to those located on the *same row* as the initial configuration; hence, it is fast but will not reach the optimal valid configuration in this example. In turn, using page migration alone can only exploit the configurations in the first column; thus, it can reach the optimal valid configuration, albeit by slow steps.

In contrast, BALM exploits both mechanisms to fix the SLO violation. This unveils the entire 2-dimensional space. BALM takes advantage of such an opportunity by exploring new configuration paths that are not available when either MBA or page migration is used alone. These new paths enable BALM to combine the virtues of each own mechanism. To illustrate this, Figure 5.3 depicts the path that BALM will follow to solve the SLO violation in our example. It is easy to see that this path heals the SLO violation as quickly as using MBA alone, while eventually reaching the valid configuration that maximizes the BEA's throughput (as page migration does).

In a nutshell, BALM finds this path in two steps. Upon detecting an SLO violation, BALM first sets MBA to its most restrictive level, to try to fix the violation as fast as possible. Next, BALM incrementally migrates pages to make the BEA (slowly) converge to the best local-to-remote ratio. Since, as each page migration step completes, the bandwidth demand on the saturated memory node is also alleviated, BALM gradually releases the MBA throttling when it observes that doing that still leaves the system in a valid configuration. Therefore, we expect BALM to (i) be as quick as MBA in fixing SLO violations, while (ii) converging to the same optimal valid configuration that the page migration will reach.

The example described so far is over-simplified, thus omitting several challenges that arise in real settings. We next describe the advanced features of BALM that address each challenge.

Uncertain and dynamic workloads. The 2-dimensional grid information in Figure 5.3 is not known a priori (neither the speedup values nor the valid configurations). Additionally, these values change dynamically as each

application's workload switches to different phases with different memory access characteristics. Hence, to find an appropriate valid configuration, BALM resorts to an online hill-climbing search, which gradually finds its way to the optimal valid configuration depicted in Figure 5.3. To accomplish this, BALM relies on a monitoring component, which continuously samples each LCA's SLO metrics and provides frequent system-wide diagnostics of the QoS health of the LCAs; namely, whether any SLO violations are already happening or prone to happen, and at which sockets they occur. Such diagnostics feed into the BALM controller, which uses such information to decide when to trigger a new reconfiguration cycle and infer the outcome of each adaptation action (enforced on a BEA).

Multiple LC applications. While the previous example considered only one LCA and one BEA, BALM needs to support QoS-aware memory bandwidth allocation in general scenarios where multiple LCAs and BEAs may be co-located at each socket of the machine. Since the system may have multiple LCAs, a bandwidth-intensive BEA may now incur SLO violations on more than one LCA. Therefore, BALM needs to collectively monitor the SLO of the ensemble of LCAs at its detection and adaptation stages. Therefore, the notion of valid configuration needs to be extended to a configuration in which *every* LCA in the system has its SLO met; otherwise, a configuration is invalid if at least one LCA's SLO is violated. Let us revisit the previous example and now assume that, instead of a single LCA, there are multiple LCAs on that same socket. It is easy to show that this redefinition of valid/invalid configuration automatically enables BALM's approach to correctly handle more general scenarios where multiple LCAs share the same socket, as long as it is the opposite socket as the BEA's.

Handling of cross-socket and intra-socket interference. In a general scenario, multiple LCAs may also be running on the same socket as the BEA. Hence, if the BEA places a high demand on the local memory, it can now (also) interfere local LCAs. In other words, besides SLO violations due to cross-socket interference (as depicted in the previous example), intra-socket interference may also trigger (local) SLO violations. BALM deals with both situations by choosing the right page migration direction when healing. Cross-socket interference is alleviated by remote-to-local migration (i.e., stepping up in Figure 5.3), whereas intra-socket interference

triggers local-to-remote migration (i.e., stepping down). Furthermore, since the LCAs may be running at different sockets, using page migration to alleviate local (resp., remote) memory pressure and fix an intra-socket (resp., cross-socket) SLO violation can have the collateral effect of introducing new cross-socket (resp., intra-socket) SLO violations.

To prevent this pathological side effect, BALM needs to ensure that such adaptation path does not enter invalid configurations (which would incur new SLO violations). BALM ensures this by collectively monitoring the SLO of every LCA after each page migration step is taken, and by rolling back to the previous (valid) configuration once a new SLO violation is found. A consequence of this measure is that, in complex scenarios such as the one illustrated above, an optimal valid configuration that totally disables MBA throttling of the BEA (i.e., MBA 100) may no longer exist.

Handling of multiple noisy neighbours. Finally, one needs to consider that more than one BEA may now simultaneously enter a bandwidth-intensive phase. Therefore, when BALM detects an SLO violation, it can aggregate the result of multiple BEAs accessing the same memory node. Hence, to fix the violation, BALM may need to adapt (via MBA and page migration) more than one BEAs. Therefore, the 2-dimensional problem described so far has its dimensionality multiplied by the number of BEAs. BALM tames this complexity by considering one BEA at a time, starting by those that, in the near past, have consumed the most memory bandwidth of the memory node where SLO violations have been detected. Intuitively, by following this heuristic, BALM tries to fix the SLO violation(s) as soon as possible by first adapting those BEAs that are most likely the root of the problem. To acquire memory bandwidth usage on a per-socket and per-application basis, BALM employs the Memory Bandwidth Monitoring (MBM) feature of the Intel RDT technology. For each BEA, the 2-dimensional online search described above is carried out. When the search completes, BALM has reached either a valid or an invalid configuration. In the former case, the SLO violations have been healed, and no more BEAs need to be adapted. In the latter case, the memory bandwidth diet imposed on the current BEA was not enough to fix the SLO violations, thus BALM moves to the next BEA in the queue.

Algorithm 4: BALM's main control loop

```
1  begin
2      retries = 0;
3      while true do
4          {color, SLOlocation} = evaluateSLO(LCA);
5          if color ∈ {yellow, red} then
               /* SLO violation(s) happening/prone to occur, so 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             end
12             for each BEA in orderedBEAs do
13                 if |SLOlocation| == 1 then
14                     if SLOlocation = BEA.socket then dir=outbound;
15                     else dir=inbound;
16                 else
17                     dir = none;
18                 end
                   /* Slowly, adapt the BEAs to find a better valid
                      configuration                                            */
19                 while (BEA.MBA< 100) do
20                     if ∈ {grey, green} then
                           /* 2a.  There is room to alleviate MBA              */
21                         color = incMBA(BEA);
22                     end
23                     if color ∈ {yellow, red} then
                           /* 2b.  Try to fix violation by migrating enough
                              pages away from victim                           */
24                         color = migratePagesUntilGrey(BEA, dir);
25                         if color ∈ {yellow, red} then
26                             break;
27                         end
28                     end
29                 end
30             end
               /* Could not fix the SLO after adapting every BEA              */
31             if color == red then
32                 if retries>MAXRETRIES then  throw
                     CannotFixSLOViolationException ;
33                 else retries++;
34             end
35             else retries=0;
36         end
37         else if color == green then
38             retries = 0;
39             if BEAs = BELoadChanged() then
40                 optimizeConfig(BEAs);
41             end
42         end
43     end
44 end
```

Algorithm

The architecture of BALM comprises a memory bandwidth allocation component and a monitoring component. The former component controls the MBA and page migration mechanisms to adjust how each BEA uses the available memory bandwidth of the multi-socket host. The latter continuously collects LCA's SLO metrics (e.g., tail latency) to detect violations and throughput of the BEAs. Regarding SLO violations, the 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. Algorithm 4 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 first ordered according to the memory bandwidth that they have recently consumed from the problematic memory node(s), in decreasing order (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 [140], 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 given BEA can take multiple iterations to converge to the ideal configuration, with no MBA restriction (MBA 100) and the optimal local-to-remote page ratio (among the valid alternatives). However, each iteration does not necessarily run both steps. For instance, in the example in Figure 5.3: a first iteration has an MBA and a page migration step; the same holds for the second iteration; however, the following iterations only run the MBA step.

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.

5.4 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 5.5). Alternatively, the application can also be instrumented to report all the necessary performance metrics such that the cloud provider has access to them (in a private cloud, internal applications are already instrumented [15]). Lastly, one can also facilitate monitoring through cloud platforms or third party applications.

Page migration mechanism. The page migration mechanism of BALM uses the approximated online page placement open source tool proposed in BWAP [140]. 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 [163]. 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.

As a final remark, we note that in architectures where MBA is unavailable, BALM can exploit any alternative bandwidth allocation mechanism (e.g.,thread packing [107] and clock modulation [108]), with similar expectations: achieve comparable SLO violation times as the baseline technique, while maximizing BEA throughput.

5.5 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?*

Experimental methodology

To answer each question, we study how BALM and other state-of-the-art alternatives handle QoS-aware memory bandwidth allocation in different dual-socket workload consolidation scenarios. 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 machines. We evaluate BALM on two dual-socket machines, a smaller machine (Machine A) with fewer cores per socket, and a bigger one (Machine B). Machine A's specifications: Intel Xeon Silver 4114 CPU, 2 NUMA nodes, 10 cores per node, 128GB DRAM (64GB per node), running Linux 4.15. Machine B's specifications: Intel Xeon Gold 5218 CPU, 2 NUMA nodes, 16 cores per node, 64GB DRAM (32GB per node), running Linux 4.19. Both support MBA, with 8 available levels.

LCA workloads. To quantify the impact of memory bandwidth interference and allocation, we consider Memcached [35] and Xapian [37] as LCAs in our experiments. Memcached is a high-performance, distributed object caching system that is mainly used to speed up web requests by caching data and objects in memory. In modern cloud services, such distributed in-memory key-value stores have become a critical tier. Several big companies, such as YouTube, Facebook, and Twitter, widely use Memcached [164].

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 on each machine 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 [15, 165]. 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 to 1ms for 99^{th} percentile latency, which is in line with the experimental deployment methodology in previous works [15, 16, 165, 166].

We use an in-house, open-loop workload generator, similar to *Mutilate* [167], as the client for Memcached. Clients run on machines on the same network as machine A/B, where Memcached runs. The load generator

uses exponential inter-arrival time distribution, similar to the query distributions at Facebook [15, 164, 168]. 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 [15, 164].

Xapian is an open-source search engine included in the Tailbench suite [54]. We used Tailbench's default configuration for Xapian. The search index is built from a snapshot of the English version of Wikipedia. We use the open-loop load generators provided by Tailbench [54]. The load generator chooses the query terms randomly, following a Zipfian distribution. This has been shown to model online search query distributions well [54]. 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 [15, 16, 166], 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 (RPS) 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 5.4 shows the relationship between tail latency and input load (RPS) for each LCA, in machine A. 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 5.4. 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 LCA instances, 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, 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 44 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 [1], PARSEC [33] and SPLASH [34]. Table 5.1 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., [16, 92]). 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. The spikes in Figure 5.2 illustrate this for *OC*. The thread count of all the evaluated applications is shown in Table 5.2.

Benchmark	Bw. requirements (GB/s		Description
	Reads	Writes	
MG.C (MG) [1]	26.31	7.16	Multi-Grid on a sequence of meshes
Ocean_cp (OC) [34]	23.81	8.48	Simulates large-scale ocean movements
SP.C (SP) [1]	20.48	10.76	Scalar Penta-diagonal solver
UA.C (UA) [1]	16.79	5.40	Unstructured Adaptive mesh, dynamic and irregular memory access
Blackscholes (BS) [33]	2.50	0.35	Option pricing with Black-Scholes Partial Diff. Equation (PDE)
EP.B (EP) [1]	0.01	0.01	Embarrassingly Parallel
Swaptions (SW) [33]	0.01	0.01	Financial analysis

Table 5.1: Evaluated BE benchmarks

Scenario	**Socket 0**	**Socket 1**
A	Memcached (4 threads)	OC/MG/SP/UA (8 threads)
B	Memcached (4 threads) + OC/MG/SP/UA (8 threads)	Xapian (4 threads) + SW (4 threads) + BS (2 threads) + EP (2 threads)

Table 5.2: The thread count of the evaluated applications

Alternative solutions. We compare BALM with *MBA* and *page migration (pgm)*, each used stand-alone; as well as the *unshared* and *unmanaged* approaches highlighted in Section 5.2. We note that state-of-the-art systems such as PARTIES [15] or CLITE [16] are large frameworks that handle

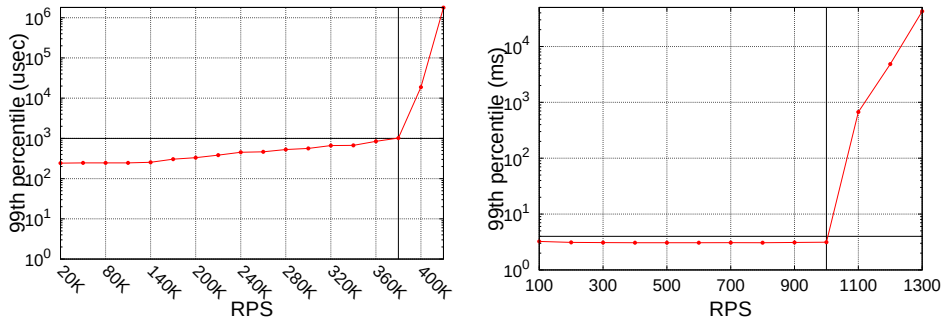


Figure 5.4: Tail latency with increasing load (RPS) of Memcached (left) and Xapian (right). The vertical line shows the *max load*, while the horizontal line shows the target SLO. The y-axis is logarithmic.

multiple shared resources. BALM should be seen as a future component of such frameworks, but not a replacement. Therefore, comparing BALM with them would not be fair, as they address broader problems. Instead, we compare BALM against the lower-level mechanisms that such frameworks use to specifically handle memory bandwidth.

Co-location Scenarios. To evaluate whether BALM achieves its goal of optimizing the performance of BEAs while safeguarding the SLO of the LCAs, we consider two co-location scenarios, which Table 5.2 summarizes. *Scenario A* takes place in the smaller machine, A. Here, we consolidate two applications only, one LCA and one BEA. These are naturally placed in opposite sockets.

In contrast, *scenario B* takes place in machine B. Having more cores available, in this scenario, we consolidate six applications. Hence, we have a larger scale, with more applications to monitor and manage; and a more 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 all 6 applications need to be monitored and managed, the main

challenge in scenario B is to address any harmful interference arising between the latter two applications.

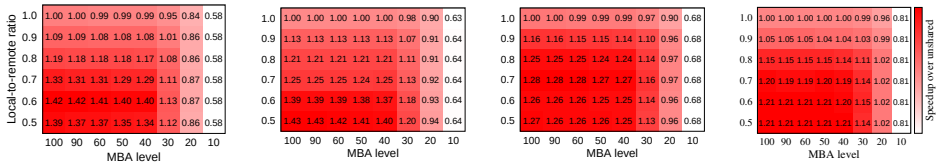
Although there are many possible application-to-socket combinations, what essentially distinguishes all of them is whether the two bandwidth-intensive applications reside at the same socket or on opposite sockets. For space limitations we only show results of scenario B for the same-socket scenario. We note that our experiments with both applications on opposite sockets yielded very similar conclusions as with scenario A.

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.

BEAs Characterization. Figure 5.5 characterizes the performance of the bandwidth-intensive BEAs when they are allocated different amounts of local-to-remote ratio[2] and MBA level. Similarly to Figure 5.3, each cell presents each BEA's throughput obtained in a given configuration, normalized to that of the system state of the *unshared* approach (i.e., the top-left cell). A first observation that stems from this overall characterization is that the performance of the benchmarks can be significantly improved by exploiting cross-socket page placement configurations, which optimize the utilization of spare memory bandwidth. Of course, due to interference with other LCAs (when these enter high-load periods), some of the cells that exhibit the best performance for each BEAs might trigger temporary SLO violations, thus should be avoided. Second, different benchmarks have distinct optimal configurations: OC, MG, and SP require a local-to-remote ratio of 0.6:0.4, 0.5:0.5, and 0.7:0.3 to achieve optimal performance.

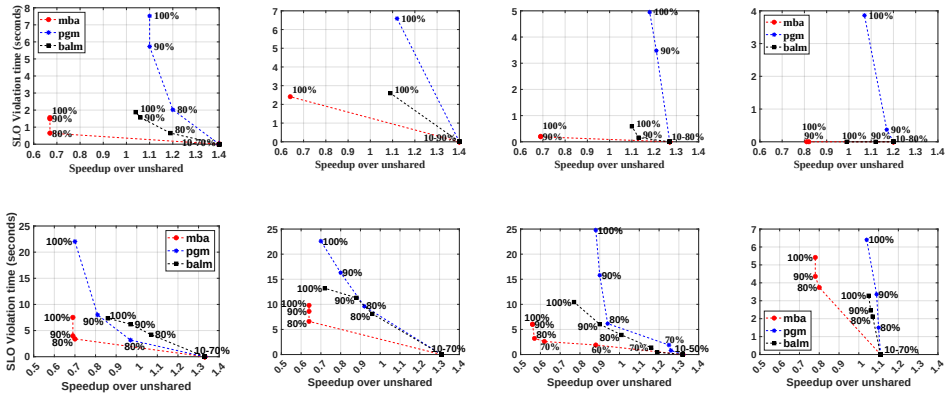
Finally, note the benchmarks achieve similar performance with different system states. For instance, UA achieves similar performance when it is allocated 0.6:0.4 local-to-remote ratio and 100% MBA level and 0.5:0.5 local-to-remote ratio and 40% MBA level.

[2]The local-to-remote ratio $l : r$ is the portion of local pages l to the portion of remote pages $r = 1 - l$. The figure shows only the portion of local pages.



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

Figure 5.5: Performance impact of page placement and MBA on BE benchmarks



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

Figure 5.6: Performance of BEAs and SLO violation time of high-load LCA for *scenario A* (top row plots) and *scenario B* (bottom row plots). 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.

Results

Scenario A. Figure 5.6 (top) presents the results for each metric (BEA performance and LCA SLO violation time) in scenario A, 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

5.6.

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 5.6 (top) 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). These trends confirm the preliminary observations from Section 5.2.

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. For instance, with OC, the SLO violation time of BALM at 80%/90%/100% (resp.) of the LCA's *max load* are just 0%/17%/5% (resp.) above *mba*'s marginal values. These results can be better understood in Figure 5.7, which details how both BALM and *mba* dynamically react to load peaks (and the corresponding SLO violations) in a specific experiment. If, instead, we consider the throughput of OC, we conclude that BALM is just 1%/4%/6% (resp.) below the performance that *pgm* achieves on that dimension. This translates to BALM outperforming *mba* and *unshared* by up to 1.78× and 1.4×, 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.

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.

To better understand the SLO violation times, Figure 5.7 shows a detailed dynamic analysis of how each solution reacts to SLO violations as time goes by. For space limitations, we only show the results for the OC BEA when the LCA is operating at 100% of its max load, in scenario A.

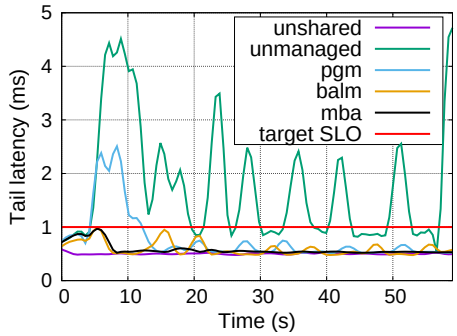


Figure 5.7: Tail latency of Memcached co-located with OC for different mechanism. Memcached is operating at 100% of its max load. The red horizontal line shows the target SLO.

Scenario B. The results of Scenario B are shown in Figure 5.6 (bottom). The main conclusions that we draw for scenario A also hold in scenario B. As before, the results show that the best-performing mechanism in terms of BEA performance are BALM and *pgm*. More precisely, BALM is able to outperform MBA approach by up to 1.87×. Additionally, both BALM and *mba* exhibit lower QoS violation times than *pgm*. The main distinctive trend of scenario B is that the time spent on page migration (by *pgm* and by the second phase of BALM) is longer. This is because QoS violations are more severe, since both the high-load LCA and bandwidth-intensive BEA are consolidated on the same socket, thus more pages need to be migrated away from the local socket. 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 5.6 (bottom) (d) is an example of the above situation.

5.6 Related Work

Recall from Section 2.3, that interference can be eliminated by partitioning resources among consolidated applications using OS- and hardware-level isolation techniques [10, 14–16, 94, 101–104]. This approach has the

following benefits: (1) it maximizes resource utilization and throughput, or trades off throughput vs. fairness [14, 57]; (2) it provides QoS for LCAs [10, 15, 16, 105]. 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.

Recent proposals [15, 16, 92] have focused on the QoS-aware resource allocation problem that is the departing point to our reformulation in Section 5.1, 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 [57] introduced a performance model to guide the use of MBA to improve performance. CoPart [14] 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 Caladan [92], BWAP [140], PARTIES [15], Heracles [10] and CLITE [16]. These systems 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 [10]. Caladan rely exclusively on core allocation to manage multiple forms of interference. However, Caladan is unable to manage interference across NUMA nodes [92]. A recent alternative has been proposed in BWAP [140], 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.

5.7 Summary

This chapter presented BALM, a QoS-aware memory bandwidth allocation technique for multi-socket architectures. The key insight of BALM is that, by combining commodity bandwidth allocation mechanisms originally designed for single-socket, with a novel adaptive cross-socket page migration scheme, we can overcome the efficiency limitations of today's state-of-the-art when deployed in multi-socket scenarios. Our experimental evaluation with real applications on different dual-socket machines shows that BALM can ensure marginal SLO violation windows while delivering substantial throughput gains to bandwidth-intensive best-effort applications, when compared to state-of-the-art alternatives.

Chapter 6

Conclusions and Future Work

In this thesis, we have worked towards optimizing the amount of Cloud resources used by applications and improving utilization of Cloud resources, by elastically adapting the resources allocated to data-intensive services and by consolidating workloads. With elastic provisioning, we aim to minimize the service provisioning cost while maintaining the desired SLOs through horizontal scaling, i.e., dynamically acquiring and releasing VMs to accommodate varying application needs. Regarding workload consolidation, we aim at improving the resource utilization of data centre and Cloud computing systems while preserving (or meeting) QoS requirements of Cloud-based services and improving the throughput of bandwidth-intensive best-effort applications in the Cloud. In order to provide and evaluate solutions for QoS-aware elasticity and resource arbitration for consolidated workloads, we have considered Cloud resource management on two levels: inter-node resource management and intra-node resource management. In the first level, we have studied how to improve the practical usefulness of an elasticity controller while deploying it in a Cloud environment, targeting distributed data stores. In resource management at the second level, i.e., the intra-node resource management, we have studied how to properly utilize over-provisioned memory resources in multi-socket hosts, to enable state-of-the-art QoS-aware resource allocation systems to be generalized to allow cross-socket sharing of memory.

In the context of inter-node resource management, to enable and achieve

QoS-aware elastic execution in the Cloud, we have designed and implemented OnlineElastMan, an "out-of-the-box" generic elasticity controller, which can be deployed and adopted by different storage systems without complex training and configuring efforts. Chapter 3 presented OnlineElastMan in detail. OnlineElastMan excels its peers with its practical aspects, including easily measurable and obtainable performance and QoS metrics, an automatically online trained control model, and an embedded generic workload prediction module. As a result, OnlineElastMan continuously improves its provision accuracy, i.e., minimizing SLO violations and provisioning cost, under various workload patterns.

In the context of intra-node resource management for efficient execution of consolidated workloads in Cloud nodes, the thesis focused on the memory bandwidth of the multi-socket NUMA system as the resource to allocate and arbitrate. To the best of our knowledge, existing proposals for memory bandwidth allocation are, by design, tailored to single-socket architectures only, which is at odds with the growing prevalence of multi-socket systems in contemporary warehouse-scale data centres. Hence, the state-of-the-art resource allocation systems need to be generalized to allow cross-socket sharing of memory. To achieve this, the low-level memory partitioning mechanisms on which existing solutions rely need to be redesigned to address the new constraints of multi-socket NUMA architectures. Two solutions, BWAP [140] and BALM [161] were designed and implemented to address these constraints.

Chapter 4 presented BWAP [140], a novel page placement mechanism based on asymmetric weighted page interleaving. BWAP is an extension of the *libnuma* library that relies on a novel combination of analytical modelling and on-line iterative tuning. Our results show that there is an unexplored opportunity in incorporating the asymmetry of NUMA topologies when placing pages of memory-intensive applications. Furthermore, to the best of our knowledge, BWAP is the first proposal for bandwidth-aware page placement in heterogeneous memory systems evaluated on real commodity machines, i.e. not by simulation [38–40].

Chapter 5 presented BALM [161], a QoS-aware memory bandwidth allocation technique for multi-socket systems. The key insight of BALM is to combine commodity bandwidth allocation mechanisms originally designed for single-socket (such as Intel's Memory Bandwidth Allocation) with a novel adaptive cross-socket page migration scheme. By doing so, BALM can overcome the limitations of the original mechanisms when deployed in

multi-socket scenarios. Our evaluation shows that, compared to state-of-the-art alternatives, BALM ensures marginal SLO for latency-sensitive applications while delivering throughput gains to best-effort applications.

In summary, this thesis has proposed methods, algorithms, and tools for improving the resource utilization of data centre and Cloud computing systems while meeting SLOs of latency-critical applications and improving the performance of best-effort applications. Specifically, a model-predictive control supported by online model training enables and achieves effective and efficient elastic execution in the Cloud, where an autonomic elastic service dynamically grows (scales out) or shrinks (scales in) as needed in order to adapt to observed or predicted changes in its workload. The elastic autoscaling of Cloud-based services with elasticity managers also allows reducing the amount of Cloud resources used by an elastic service, and as a consequence, reducing the cost of the used Cloud resources and improving Cloud resource utilization. Additionally, QoS-aware memory bandwidth partitioning techniques can ensure marginal SLO violation windows while delivering better performance for bandwidth-intensive best-effort applications running on multi-socket Cloud nodes. We have addressed the research questions presented in this thesis with novel solutions, and all the proposed solutions, namely OnlineElastMan, BWAP and BALM, have been implemented and evaluated on a limited set of real-world workloads. Our experimental evaluation indicates the feasibility and effectiveness of our proposed approaches to inter-node resource and intra-node resource management aimed at improving resource utilization through QoS-aware elastic execution in the Cloud and effective arbitration of resources among consolidated workloads in Cloud nodes.

6.1 Future Work

In future, the research works presented in this thesis can be extended in several dimensions as described below. We present them below, organized according to the main contributions of this thesis.

- OnlineElastMan:
  - First, it would be useful to extend the control model of OnlineElastMan with more comprehensive metrics, e.g., CPU utilization, network statistics, disk I/Os, etc.

  - Second, OnlineElastMan, which is inherently centralized, can be decentralized for better scalability and fault tolerance. OnlineElastMan is essentially stateless, as states are only preserved and used in the prediction and model training modules, which can be generated/trained during runtime. Thus, we envision decentralization to be technically viable without fundamental changes to the underlying algorithms.

- BWAP:

  - **Account for workloads with relevant write and/or thread-private access volumes.** As discussed in section 4.3, we assume that the canonical application is mostly read-only regarding shared data. However, this is not true for most real world applications, as shown in Table 4.1. Additionally, we assume a negligible impact of thread-private pages on the overall memory throughput. Therefore, BWAP can be generalized to account for these factors. Ideally, the profiled throughput between each pair of nodes should consist of the read, write and local bandwidths. Profiling and including these extra parameters into the `Canonical tuner` equations is currently an open problem. To handle these and additional factors with BWAP, the techniques of previous work e.g., Integer Programming (IP) [112] can be incorporated into `Canonical tuner` in the future. NuCore [112] has shown that employing IP achieves high accuracy and low overhead.

  - **Account for workloads with non-uniform access distributions to the shared address space.** BWAP also assumes that an application accesses all shared pages with the same probability. However, some applications have skewed access distributions, where a large portion of the application's footprint is infrequently accessed [169]. Therefore, our mechanism need to distinguish frequently accessed pages (hot) from infrequently accessed ones (cold) for such applications to improve their performance. Some existing mechanisms take advantage of the `Accessed bit` in the page table entry (PTE) which is set by the hardware each time the PTE is accessed [170]. Others rely on hardware counters (sampling) to gather page accesses statistics [12]. Detecting hot pages

is an area of ongoing research [12, 39, 170] which requires monitoring at high frequency, resulting in unacceptable slowdowns. Hence, it might be worthwhile to investigate methods to detect hot pages and support mapping of these pages to high bandwidth links without significantly impacting performance. Consequently, rarely accessed pages can be mapped to low bandwidth links.

  - **Heterogeneous memory subsystems.** BWAP can also be extended to support NUMA systems whose nodes have hybrid memory subsystems (e.g. DRAM and NVRAM). Recent works have studied the problem of page placement on emerging hybrid memory hierarchies [39, 40]. These works usually consider a single-node system, where the physical address space is partitioned across different memory technologies with heterogeneous performance characteristics, such as bandwidth, latency, write-endurance and persistence. While page placement in this context is a fundamentally different problem than the one that we address, they clearly complement each other. In that sense, BWAP can be combined with recent solutions for single-node hybrid memory hierarchies.

- BALM:

  - First, BALM can be enhanced with extra features based on the use of available hardware technologies, in order to support 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 actual effectiveness of BALM still needs to be experimentally evaluated in such settings.

Bibliography

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[4] Google Compute Engine. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling.

[5] Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi. A design methodology for self-management in distributed environments. In *2009 International Conference on Computational Science and Engineering*, volume 1, pages 430–436. IEEE, 2009.

[6] Are noisy neighbors in your data center keeping you up at night? https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-rdt-infrastructure-paper.pdf. Accessed: 2021-03-29.

[7] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience.

In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–197, 2013.

[8] K. Sudan, S. Srinivasan, R. Balasubramonian, and R. Iyer. Optimizing datacenter power with memory system levers for guaranteed quality-of-service. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 117–126, 2012.

[9] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[10] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[11] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 277–289, Berkeley, CA, USA, 2015. USENIX Association.

[12] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the*

27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 10:1–10:16, New York, NY, USA, 2019. ACM.

[15] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 107–120, New York, NY, USA, 2019. ACM.

[16] T. Patel and D. Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020.

[17] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[18] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[19] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[20] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and*

Distributed Computing, HPDC '13, pages 115–116, New York, NY, USA, 2013. ACM.

[21] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 7:1–7:10, New York, NY, USA, 2013. ACM.

[22] M Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *The 8th International Conference on Autonomic and Autonomous Systems (ICAS 2012)*, pages 589–596, 2012.

[23] Ying Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 453–464, May 2015.

[24] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Hubbub-scale: Towards reliable elastic scaling under multi-tenancy. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 233–244, 2016.

[25] M. Moulavi, A. Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *ICAS 2012*, 2012.

[26] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. *Stay-Away*, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, page 301–312, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Ying Liu, Daharewa Gureya, Ahmad Al-Shishtawy, and Vladimir Vlassov. Onlineelastman: Self-trained proactive elasticity manager for cloud-based storage services. *Cluster Computing*, 20(3):1977–1994, September 2017.

[28] Intel resource director technology. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. Accessed: 2020-10-09.

[29] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 283–294, 2011.

[30] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[31] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 33–48, Berkeley, CA, USA, 2016. USENIX Association.

[32] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris, and Vasileios Trigonakis. Abstracting multi-core topologies with mctop. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 544–559, New York, NY, USA, 2017. ACM.

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

[34] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

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

[36] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

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

[38] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 268–280, New York, NY, USA, 2017. ACM.

[39] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. *SIGPLAN Not.*, 50(4):607–618, March 2015.

[40] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 18:1–18:10, New York, NY, USA, 2017. ACM.

[41] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[42] G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270, Nov 2012.

[43] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[44] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018.

[45] C. Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[46] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS

'20, page 483–495, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 58–65, 2013.

[48] Moriyoshi Ohara, Priya Nagpurkar, Yohei Ueda, and Kazuaki Ishizaki. The data-centricity of web 2.0 workloads and its impact on server performance. In *ISPASS*, pages 133–142. IEEE, 2009.

[49] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS director: scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 12–12, USA, 2011.

[50] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[51] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *The 10th USENIX Conference on File and Storage Technologies (FAST'12)*, February 2012.

[52] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[53] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[54] H. Kasture and D. Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE In-*

ternational Symposium on Workload Characterization (IISWC), pages 1–10, 2016.

[55] Christina Delimitrou and Christos 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*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery.

[56] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 519–531, USA, 2018. USENIX Association.

[57] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 16:1–16:12, New York, NY, USA, 2019. ACM.

[58] Y. Guo, A. L. Stolyar, and A. Walid. Online vm auto-scaling algorithms for application hosting in a cloud. *IEEE Transactions on Cloud Computing*, 8(3):889–898, 2020.

[59] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX.

[60] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

[61] Danny Yuan, Neeraj Joshi, Daniel Jacobson, Puneet Oberai. Scryer: Netflix's Predictive Auto Scaling Engine. http://techblog.netflix.com/2013/12/scryer-netflixs-predictive-auto-scaling.html. [Online; accessed June 2015].

[62] Right Scale. http://www.rightscale.com/.

[63] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[64] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 95–104, New York, NY, USA, 2014. ACM.

[65] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, December 2014.

[66] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.

[67] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. From data center resource allocation to control theory and back. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 410–417, 2010.

[68] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: A reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, page 137–146, New York, NY, USA, 2009. Association for Computing Machinery.

[69] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 IEEE International Conference on Autonomic Computing*, pages 65–73, 2006.

[70] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In

2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pages 64–73, 2017.

[71] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Gener. Comput. Syst.*, 32(C):82–98, March 2014.

[72] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1), March 2008.

[73] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *The 13th IEEE/IFIP Network Operations and Management Symposium*, NOMS 2012, Hawaii, USA, April 2012.

[74] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Towards faster response time models for vertical elasticity. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 560–565, 2014.

[75] Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 1000 islands: Integrated capacity and workload management for the next generation data center. In *2008 International Conference on Autonomic Computing*, pages 172–181, 2008.

[76] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010.

[77] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):800–813, 2019.

[78] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 43–50, 2007.

[79] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[80] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, Oct 2010.

[81] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Bayuh Lakew, Ivona Brandic, and Erik Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65:57 – 72, 2016. Special Issue on Big Data in the Cloud.

[82] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth. Coordinating cpu and memory elasticity controllers to meet service response time constraints. In *2015 International Conference on Cloud and Autonomic Computing*, pages 69–80, 2015.

[83] Azure CloudMonix. https://cloudmonix.com/.

[84] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale: Automatic application scaling in enterprise clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 221–228, 2012.

[85] Horizontal pod autoscaler. https://v1-17.docs.kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2021-03-04.

[86] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Augmenting elasticity controllers for improved accuracy. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 117–126, 2016.

[87] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, September 2004.

[88] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology, October 15 2001.

[89] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.

[90] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[91] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179, 2020.

[92] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[93] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou 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*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[94] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 301–312. IEEE Press, 2014.

[95] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 351–364, New York, NY, USA, 2013. Association for Computing Machinery.

[96] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[97] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

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

[99] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 97–110, New York, NY, USA, 2015. Association for Computing Machinery.

[100] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 619–630, New York, NY, USA, 2013. Association for Computing Machinery.

[101] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 729–742, New York, NY, USA, 2014. Association for Computing Machinery.

[103] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 598–610, New York, NY, USA, 2015. Association for Computing Machinery.

[104] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, page 57–68, New York, NY, USA, 2011. Association for Computing Machinery.

[105] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. Dicer: Diligent cache partitioning for efficient workload consolidation. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[106] X. Wang, S. Chen, J. Setter, and J. F. Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, Feb 2017.

[107] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, 2011.

[108] P. Cicotti, A. Tiwari, and L. Carrington. Efficient speed (es): Adaptive dvfs and clock modulation for energy efficiency. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 158–166, 2014.

[109] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 479–488, New York, NY, USA, 2009. Association for Computing Machinery.

[110] D. R. Hower, H. W. Cain, and C. A. Waldspurger. Pabst: Proportionally allocated bandwidth at the source and target. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 505–516, 2017.

[111] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, page 25–36, New York, NY, USA, 2007. Association for Computing Machinery.

[112] W. Wang, J. W. Davidson, and M. L. Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431, March 2016.

[113] M. Diener, E. H. M. Cruz, M. A. Z. Alves, P. O. A. Navaux, A. Busse, and H. U. Heiss. Kernel-based thread and data mapping for improved memory affinity. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2653–2666, Sept 2016.

[114] Zoltan Majo and Thomas R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, June 2011.

[115] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.

[116] Daniel Goodman, Georgios Varisteas, and Tim Harris. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of*

the Twelfth European Conference on Computer Systems, EuroSys '17, page 254–269, New York, NY, USA, 2017. Association for Computing Machinery.

[117] Jonathan Corbet. Toward better numa scheduling, March 2012.

[118] Amin Mohtasham and João Barreto. Rubic: Online parallelism tuning for co-located transactional memory applications. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 99–108, New York, NY, USA, 2016. ACM.

[119] Giorgis Georgakoudis, Hans Vandierendonck, Peter Thoman, Bronis R. De Supinski, Thomas Fahringer, and Dimitrios S. Nikolopoulos. Scalo: Scalability-aware parallelism orchestration for multi-threaded workloads. *ACM Trans. Archit. Code Optim.*, 14(4):54:1–54:25, December 2017.

[120] numa - numa policy library. https://linux.die.net/man/3/numa. Accessed: 2018-01-10.

[121] Jonathan Corbet. Autonuma: the other approach to numa scheduling, March 2012.

[122] Aws cassandra: Cassandra, numa and ec2. http://cloudurable.com/blog/cassadra-numa-ec2-aws/index.html. Accessed: 2018-01-10.

[123] The mongodb 4.0 manual. https://docs.mongodb.com/manual/administration/production-notes/. Accessed: 2018-01-10.

[124] Mysql bugs, bug #57241. https://bugs.mysql.com/bug.php?id=57241. Accessed: 2018-01-10.

[125] Numa support in windows. https://docs.microsoft.com/en-gb/windows/win32/procthread/numa-support?redirectedfrom=MSDN. Accessed: 2021-06-23.

[126] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, USA, 2006.

[127] Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. Modeling virtual machine performance: challenges and approaches. *SIGMETRICS Perform. Eval. Rev.*, 37(3):55–60, January 2010.

[128] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. VM3: Measuring, modeling and managing VM shared resources. *Computer Networks*, 53(17):2873–2887, December 2009.

[129] DataStax, Inc. Apache Cassandra 1.2 DATASTAX Documentation. http://docs.datastax.com/en. [Online; accessed June 2015].

[130] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 219–230, USA, 2013. USENIX Association.

[131] Steve R. Gunn. Support vector machines for classification and regression. Technical report, University of Southampton, 1998.

[132] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.

[133] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[134] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *Network, IEEE*, 14(3):30–37, May 2000.

[135] R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *Selected Areas in Communications, IEEE Journal on*, 9(2):203–211, Feb 1991.

[136] Wei-Yin Loh. Classification and regression trees. *WIREs Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[137] Robert Nau. Statistical forecasting: notes on regression and time series analysis. http://people.duke.edu/~rnau/411home.htm. [Online; accessed June 2015].

[138] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[139] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, February 1994.

[140] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov. Bandwidth-aware page placement in numa. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 546–556, 2020.

[141] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Coscheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 24:1–24:14, New York, NY, USA, 2014. ACM.

[142] G. Chatzopoulos et al. ESTIMA: Extrapolating scalability of in-memory applications. *ACM Trans. Parallel Comput.*, 4(2):10:1–10:28, 2017.

[143] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa. Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 380–391, Feb 2014.

[144] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa. Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model. In Stephen Jarvis, Steven Wright, and Simon Hammond, editors, *High Performance Computing systems - Performance Modeling, Benchmarking, and Simulation - 8th International Workshop, PMBS 2017*, volume 10724 of *Lecture Notes in Computer Science*, pages 91–113, Denver (CO), United States, November 2017. Springer.

[145] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, 2010.

[146] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. Numamma: Numa memory analyzer. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[147] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, Philadelphia, PA, June 2014. USENIX Association.

[148] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 263–276, Berkeley, CA, USA, 2015. USENIX Association.

[149] Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. Analytics with smart arrays: Adaptive and efficient language-independent data. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 17:1–17:15, New York, NY, USA, 2018. ACM.

[150] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. Memprof: A memory profiler for numa multicore systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[151] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. Lira: Adaptive contention-aware thread placement for parallel runtime systems. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.

[152] Zoltan Majo and Thomas R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 12:1–12:10, New York, NY, USA, 2011. ACM.

[153] Zoltan Majo and Thomas R. Gross. Matching memory access patterns and data placement for numa systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 230–241, New York, NY, USA, 2012. ACM.

[154] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proceedings*

of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), HPCA '13, pages 306–317, Washington, DC, USA, 2013. IEEE Computer Society.

[155] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Tumbler: An effective load-balancing technique for multi-cpu multicore systems. *ACM Trans. Archit. Code Optim.*, 12(4):36:1–36:24, November 2015.

[156] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Coherence stalls or latency tolerance: Informed cpu scheduling for socket and core sharing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 323–336, Berkeley, CA, USA, 2016. USENIX Association.

[157] Tim Harris and Stefan Kaestle. Callisto-rts: Fine-grain parallel loops. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 45–56, Berkeley, CA, USA, 2015. USENIX Association.

[158] Zoltan Majo and Thomas R. Gross. A library for portable and composable data locality optimizations for numa systems. *ACM Trans. Parallel Comput.*, 3(4):20:1–20:32, March 2017.

[159] Lei Liu, Hao Yang, Yong Li, Mengyao Xie, Lian Li, and Chenggang Wu. Memos: A full hierarchy hybrid memory management framework. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 368–371, 2016.

[160] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. Concurrent migration of multiple pages in software-managed hybrid main memory. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 420–423, 2016.

[161] David Gureya, Vladimir Vlassov, and João Barreto. Balm: Qos-aware memory bandwidth partitioning for multi-socket cloud nodes. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 435–438, New York, NY, USA, 2021. Association for Computing Machinery.

[162] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

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

[164] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[165] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez. Workload characterization of interactive cloud services on big and small server platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134, 2017.

[166] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *SIGOPS Oper. Syst. Rev.*, 50(2):33–47, March 2016.

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

[168] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[169] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[170] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGARCH Comput. Archit. News*, 45(1):631–644, April 2017.