UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Interference Analysis and Resource Management in Server Processors: From HPC to Cloud Computing

A thesis submitted in fulfillment with the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Author

Lucía Pons Escat

Advisor

Prof. Julio Sahuquillo Borrás

Valencia, June 2023

Agradecimientos

*"**El éxito** es la consecuencia directa de la **perseverancia**".*
*"**La gratitud nos** fortalece, enriquece y **eleva**". Rafa Sansores Majul*

Esta tesis es la culminación de años de esfuerzo, dedicación y determinación: un sueño cumplido. El doctorado ha supuesto una experiencia de aprendizaje excepcional. No ha sido un camino fácil: una pandemia, una diabetes y muchos resultados fallidos, pero la constancia y la perseverancia han sido clave para superar todos los obstáculos y alcanzar mis metas. Por ello, escribo estas palabras muy feliz y orgullosa de mi misma.

Una parte fundamental para haber logrado esta tesis han sido todas las personas que han estado junto a mi, cuyo apoyo ha sido crucial. Quiero agradecer, en primer lugar, a mi director de tesis, Julio Sahuquillo, quien ha confiado en mi desde que era alumna de 2ndo de grado y me ha guiado en mi carrera investigadora. Gracias por todo lo que me has enseñado y las innumerables horas que has dedicado a ayudarme a realizar los trabajos de esta tesis. Gracias por involucrarme en diferentes proyectos que, aunque a veces fuera todo un reto, me han permitido adquirir una experiencia de valor incalculable.

En segundo lugar, quisiera agradecer a las personas con las que he tenido el placer de trabajar. A Timothy M. Jones, por su gran acogida en la Universidad de Cambridge (a pesar de la pandemia) así como su contribución en uno de los trabajos realizados en esta tesis. A Chaoyi Huang, por proponernos un proyecto ambicioso del cual es fruto una parte importante de esta tesis. A Salva Petit y María Engracia Gómez, por su ayuda y sus valiosos consejos que han enriquecido los trabajos de esta tesis. A mi padre, Julio Pons, por ser mi mano derecha y ayudarme a resolver los problemas que surgían. A Vicent Selfa, por iniciarme en los primeros pasos y obligarme a dar lo mejor de mi. A Josué Feliu, por compartir su experiencia y ayudarme con las dudas que he tenido. A Marta Navarro, por ayudarme a desarrollar la faceta de enseñar y ser una gran compañera.

También quiero dar las gracias a mis compañeros de laboratorio y del GAP por los buenos momentos que hemos disfrutado juntos, sobre todo en los viajes a ACACES, SARTECO 2022 y Nápoles. Y por supuesto a mis amigos/as y a mi falla por esos momentos de desconexión y risas tan necesarios.

Finalmente, quiero agradecer a las personas más importantes de mi vida, mi familia. A mis padres, gracias por inculcarme el valor del esfuerzo y enseñarme a sobreponerme a las adversidades. Pero sobretodo, por apoyarme en cada paso que doy y animarme en los momentos más difíciles. Vuestra sonrisa de orgullo es la mejor recompensa. A mis abuelos, por su amor infinito que siempre me da fuerzas. A mis hermanos mayores, con los que puedo contar siempre. A Julio y Miljana, por su apoyo y cariño, y al pequeño Julio que ha llenado la familia de felicidad. A Alfonso, por ser mi fiel compañero y estar siempre a mi lado. Y también a mi novio, Mario, por sacarme una sonrisa cuando más lo necesito. Gracias por motivarme a ser mejor cada día y perseguir mis sueños.

Abstract

One of the main concerns of today's data centers is to maximize server utilization. In each multi-core server processor, multiple applications are executed concurrently, increasing resource efficiency as system resources are shared. However, performance and fairness are highly dependent on the share of resources that each application receives, leading to performance unpredictability. The rising number of cores (and running applications) with every new generation of processors is leading to a growing concern for interference at the shared resources.

Resource sharing has been typically tackled in High-Performance Computing (HPC); nevertheless, due to the increasing prominence of cloud computing, issues typically handled in HPC are now being targeted in this domain. This thesis focuses on addressing resource interference when different applications are consolidated on the same server processor from two main perspectives: HPC and cloud computing.

In the context of HPC, resource management approaches are proposed to reduce inter-application interference at two major critical resources: the last level cache (LLC) and the processor cores. The LLC plays a key role in the system performance of current multi-cores by reducing the number of long-latency main memory accesses. LLC partitioning approaches are proposed for both inclusive and non-inclusive LLC, as both designs are present in current server processors. In both cases, newly *problematic* LLC behaviors are identified and efficiently detected, granting a larger cache share to those applications that make profitable use of the LLC space. As for processor cores, many parallel applications, like graph applications, do not scale well with an increasing number of threads/processes due to hardware and software issues. However, the default Linux time-sharing scheduler performs poorly when running graph applications, which, unlike other scientific applications, process vast amounts of data. To maximize system utilization, this thesis proposes to co-locate multiple graph applications on the same server processor by assigning the optimal number of cores to each one. By dynamically adapting the number of threads spawned by the running applications, it is possible to change the number of cores allocated to meet applications' runtime requirements.

When studying the impact of system shared resources on cloud computing, this thesis addresses three major challenges: the complex infrastructure of cloud systems, the nature of cloud applications, and the impact of inter-VM interference on the VMs' performance. Firstly, this dissertation presents the experimental platform developed to perform representative cloud performance studies with the main cloud system components (hardware and software). Secondly, an extensive characterization study is presented on a set of representative latency-critical workloads as many important cloud workloads must meet strict quality of service (QoS) requirements to provide a satisfactory user experience. The aim of the studies is to outline considerations cloud providers should take into account to improve performance and resource utilization. Finally, we propose an online approach that detects and accurately estimates inter-VM interference in scenarios with multiple co-located latency-critical VMs. The approach relies on metrics that can be easily monitored in the public cloud as VMs are handled as "black boxes". The research described above is carried out following the restrictions and requirements to be applicable to public cloud production systems.

In summary, this thesis addresses contention in the main system shared resources in the context of server consolidation, both in HPC and cloud computing. Experimental results show that important gains are obtained over the Linux OS scheduler by reducing the interference at the shared resources. In inclusive LLCs, turnaround time (TT) is reduced by over 40% while sustaining (and even improving) IPC by more than 3%. In non-inclusive LLCs, fairness and TT are improved by 44% and 24%, respectively, while even improving performance up to 3.5%. By distributing core resources efficiently, an almost perfect fairness can be obtained (on average 94%), and TT can be reduced by up to 80% In cloud computing environments, performance degradation due to resource contention can be estimated with an overall prediction error of 5%. All of the approaches proposed in this dissertation have been designed to be applied in commercial server processors without requiring any prior information. Decisions are performed dynamically at execution time using the data collected from the hardware performance counters.

Resumen

Una de las principales preocupaciones de los centros de datos actuales es maximizar la utilización de los servidores. En cada servidor o procesador multinúcleo se ejecutan simultáneamente varias aplicaciones, lo que aumenta la eficiencia de los recursos gracias a la compartición de los mismos. Sin embargo, el rendimiento y la equidad dependen en gran medida de la proporción de recursos que recibe cada aplicación, lo que provoca que su tiempo de ejecución sea imprevisible. El creciente número de núcleos (y de aplicaciones ejecutándose al mismo tiempo) con cada nueva generación de procesadores hace que crezca la preocupación por el efecto causado por las interferencias en los recursos compartidos.

La compartición de recursos se ha abordado típicamente en la computación de alto rendimiento (HPC); sin embargo, debido a la creciente importancia de la computación en la nube, los problemas que suelen tratarse en HPC se han trasladado a este ámbito. Esta tesis se centra en mitigar la interferencia en los recursos compartidos cuando diferentes aplicaciones se consolidan en un mismo procesador desde dos perspectivas: HPC y computación en la nube.

En el contexto de HPC, para reducir la interferencia causada por la ejecución concurrente de múltiples aplicaciones, en esta tesis se proponen políticas de gestión para dos de los recursos más críticos del sistema: la caché de último nivel (LLC) y los núcleos del procesador. La LLC desempeña un papel clave en las prestaciones del sistema con los procesadores multinúcleo actuales ya que reducen considerablemente el número de accesos de alta latencia a la memoria principal. Se proponen estrategias de particionado de la LLC tanto para cachés inclusivas como no inclusivas, ya que ambos diseños están presentes en la actualidad en los procesadores para servidores. Para los dos esquemas de caché, se identifican y detectan eficientemente nuevos comportamientos problemáticos en lo que se refiere a la LLC. Esto permite asignar un mayor espacio de caché a aquellas aplicaciones que hacen un uso eficiente del mismo. En cuanto a los núcleos del procesador, muchas aplicaciones paralelas, como las aplicaciones de grafos, no escalan bien a medida que se incrementa el número de hilos/procesos debido a problemas *hardware* y *software*. Sin embargo, el planificador de Linux, que aplica una estrategia de tiempo compartido, no ofrece buenas prestaciones cuando se ejecutan aplicaciones de grafo, ya que, a diferencia de otras aplicaciones científicas, procesan grandes cantidades de datos. Para maximizar la utilización del sistema, esta tesis propone ejecutar múltiples aplicaciones de grafo simultáneamente en el mismo procesador, asignando el número óptimo de núcleos a cada una. Adaptando el número de hilos creados por las aplicaciones en tiempo de ejecución, es posible cambiar el número de núcleos asignados para satisfacer los requisitos de las aplicaciones de manera dinámica.

Para estudiar el impacto de los recursos compartidos del sistema en la computación en la nube, esta tesis aborda tres grandes retos: la compleja infraestructura de los sistemas en la nube, las características de las aplicaciones que se ejecutan en la nube y el impacto de la interferencia entre máquinas virtuales (MV) en el rendimiento de éstas. En primer lugar, esta tesis presenta la plataforma experimental desarrollada con los principales componentes de un sistema en la nube (*hardware* y *software*) para realizar estudios representativos del rendimiento de la nube. En segundo lugar, se presenta un amplio estudio de caracterización sobre un conjunto de aplicaciones de latencia crítica representativas, ya que muchas cargas de trabajo importantes en la nube deben cumplir estrictos requisitos de calidad de servicio (QoS) para brindar una experiencia de usuario satisfactoria. El objetivo de los estudios es identificar las cuestiones que los proveedores de servicios en la nube deben tener en cuenta para mejorar el rendimiento y la utilización de los recursos. Por último, se realiza una propuesta que, de manera dinámica, permite detectar y estimar de forma precisa la interferencia entre MV en escenarios en los que se ejecutan múltiples MV con aplicaciones de latencia crítica. El enfoque se basa en métricas que pueden monitorizarse fácilmente en la nube pública, ya que las MV deben tratarse como "cajas negras". Toda la investigación descrita se lleva a cabo respetando las restricciones y cumpliendo los requisitos para ser aplicable en entornos de producción en la nube pública.

En resumen, esta tesis aborda la contención en los principales recursos compartidos del sistema en el contexto de la consolidación de servidores, tanto en entornos de altas prestaciones como en entornos de nube. Los resultados experimentales muestran importantes ganancias de prestaciones sobre el planificador del sistema operativo Linux al reducir las interferencias en los recursos compartidos. En los procesadores con LLC inclusiva, el tiempo de ejecución (TT) se reduce en más de un 40 %, mientras que se mantiene (e incluso mejora) el IPC en más de un 3 %. En los sistemas con LLC no inclusiva, la equidad y el TT mejoran en un 44 % y un 24 %, respectivamente, al mismo tiempo que se obtiene una mejora del rendimiento de hasta en un 3,5 %. Al distribuir los núcleos del procesador de forma eficiente, se alcanza una equidad casi perfecta (de media un 94 %), y el TT puede reducirse hasta un 80 %. En entornos de computación en la nube, la degradación del rendimiento debido a la contención en los recursos compartidos puede estimarse con un error de un 5 % en la predicción global. Todas las propuestas presentadas en esta tesis han sido diseñadas para ser aplicadas en procesadores de servidores comerciales sin requerir ninguna información previa. Las decisiones se toman dinámicamente en tiempo de ejecución utilizando los datos recogidos de los contadores de prestaciones *hardware*.

Resum

Una de les principals preocupacions dels centres de dades actuals és maximitzar la utilització dels servidors. A cada servidor o processador multinucli s'executen simultàniament diverses aplicacions, cosa que augmenta l'eficiència dels recursos gràcies a la compartició dels mateixos. Tot i això, el rendiment i l'equitat depenen en gran mesura de la proporció de recursos que rep cada aplicació, cosa que provoca que el seu temps d'execució siga imprevisible. El nombre creixent de nuclis (i aplicacions executant-se alhora) amb cada nova generació de processadors fa que creixca la preocupació per l'efecte causat per les interferències en els recursos compartits.

La compartició de recursos s'ha abordat típicament a la computació d'alt rendiment (HPC); no obstant això, a causa de la creixent importància de la computació al núvol, els problemes que solen tractar-se en HPC s'han traslladat a aquest àmbit. Aquesta tesi se centra a mitigar la interferència en els recursos compartits quan diferents aplicacions es consoliden en un mateix processador des de dues perspectives: HPC i computació al núvol.

En el context d'HPC, per reduir la interferència causada per l'execució concurrent de múltiples aplicacions, en aquesta tesi es proposen polítiques de gestió per a dos dels recursos més crítics del sistema: la memòria cau d'últim nivell (LLC) i els nuclis del processador. La LLC exerceix un paper clau a les prestacions del sistema en els processadors multinucli actuals ja que redueixen considerablement el nombre d'accessos d'alta latència a la memòria principal. Es proposen estratègies de particionament de la LLC tant per a caus inclusives com no inclusives, ja que ambdós dissenys són presents actualment en els processadors per a servidors. Per als dos esquemes de memòria cau, s'identifiquen i detecten eficientment nous comportaments problemàtics pel que fa a la LLC. Això permet assignar un major espai de memòria cau a aquelles aplicacions que en fan un ús eficient. Pel que fa als nuclis del processador, moltes aplicacions paral·leles, com les aplicacions de grafs, no escalen bé a mesura que s'incrementa el nombre de fils/processos a causa de problemes *hardware* i *software*. Tot i això, el planificador de Linux, que aplica una estratègia de temps compartit, no ofereix bones prestacions quan s'executen aplicacions de graf, ja que, a diferència d'altres aplicacions científiques, processen grans quantitats de dades. Per maximitzar la utilització del sistema, aquesta tesi proposa executar múltiples aplicacions de grafs simultàniament al mateix processador, assignant el nombre òptim de nuclis a cadascuna. Adaptant el nombre de fils creats per les aplicacions en temps d'execució, és possible canviar el nombre de nuclis assignats per satisfer els requisits de les aplicacions de manera dinàmica.

Per estudiar l'impacte dels recursos compartits del sistema a la computació al núvol, aquesta tesi aborda tres grans reptes: la complexa infraestructura dels sistemes al núvol, les característiques de les aplicacions que s'executen al núvol i l'impacte de la interferència entre màquines virtuals (MV) al rendiment d'aquestes. En primer lloc, aquesta tesi presenta la plataforma experimental desenvolupada amb els principals components d'un sistema al núvol (*hardware* i *software*) per fer estudis representatius del rendiment del núvol. En segon lloc, es presenta un ampli estudi de caracterització sobre un conjunt d'aplicacions de latència crítica representatives, ja que moltes càrregues de treball importants al núvol han de complir requisits de qualitat de servei estrictes (QoS) per brindar una experiència d'usuari satisfactòria. L'objectiu dels estudis és identificar les qüestions que els proveïdors de serveis al núvol han de tenir en compte per millorar el rendiment i la utilització dels recursos. Finalment, es fa una proposta que de manera dinàmica permet detectar i estimar de manera precisa la interferència entre MV en escenaris on s'executen múltiples MV amb aplicacions de latència crítica. L'enfocament es basa en mètriques que es poden monitoritzar fàcilment al núvol públic, ja que les MV han de tractar-se com a "caixes negres". Tota la investigació descrita es duu a terme respectant les restriccions i complint els requisits per ser aplicable en entorns de producció al núvol públic.

En resum, aquesta tesi aborda la contenció en els principals recursos compartits del sistema en el context de la consolidació de servidors, tant en entorns d'altes prestacions com en entorns de núvol. Els resultats experimentals mostren que s'obtenen importants guanys sobre el planificador del sistema operatiu Linux en reduir les interferències en els recursos compartits. En els processadors amb una LLC inclusiva, el temps d'execució (TT) es redueix en més d'un 40%, mentres que es manté (i fins i tot millora) l'IPC en més d'un 3%. En els que tenen una LLC no inclusiva, l'equitat i el TT es milloren en un 44% i un 24%, respectivament, al mateix temps que s'obté una millora del rendiment de fins a un 3,5%. Distribuint els nuclis del processador de manera eficient es pot obtindre una equitat quasi perfecta (de mitjana un 94%), i el TT pot reduir-se fins a un 80%. En entorns de computació al núvol, la degradació del rendiment degut a la contenció en els recursos compartits pot estimar-se amb un error de predicció global d'un 5%. Totes les propostes presentades en aquesta tesi han sigut dissenyades per a ser aplicades en processadors de servidors comercials sense requerir cap informació prèvia. Les decisions es prenen dinàmicament en temps d'execució utilitzant les dades recollides dels comptadors de prestacions *hardware*.

List of Acronyms

ANTT	Average Normalized Turnaround Time
CBM	Capacity Bitmask
CLOS	Class of Service
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random Access Memory
EDP	Energy Delay Product
FaaS	Function as a Service
GIPS	Giga Instructions Per Second
HPC	High-Performance Computing
HPKI_LLC	Hits Per Kilo Instruction of the LLC
I/O	Input/Output
IaaS	Infrastructure as a Service
Intel CAT	Intel Cache Allocation Technology
Intel MBA	Intel Memory Bandwidth Allocation
Intel RDT	Intel Resource Director Technologies
IPC	Instructions Per Cycle
LLC	Last Level Cache
MPKC_LLC	Misses Per Kilo Cycle of the LLC
MPKI_LLC	Misses Per Kilo Instruction of the LLC
NUMA	Non-Uniform Memory Access
PCIe	Peripheral Component Interconnect Express
PID	Process Identifier
QoS	Quality of Service
QPS	Queries Per Second
SLA	Service Level Agreement
SMT	Simultaneous MultiThreading
SSD	Solid-State Drives
TT	Turnaround Time
vCPU	Virtual CPU
VM	Virtual Machine

Contents

CHAPTER 1

Introduction

This chapter presents the main research issues addressed in this doctoral thesis and the motivation for the work done. To help contextualize the research problem, first, important background information is provided on interference at the main system resources, focusing on the last level cache (LLC) and core resources. Resource sharing has been typically addressed in High-Performance Computing (HPC); however, the growing popularity of cloud computing makes that problems commonly faced in HPC shift to this area. In this regard, compared to HPC, the challenges faced in cloud computing are magnified due to the importance of managing resource interference in public clouds to meet QoS requirements. Finally, the objectives and main contributions of this thesis are described.

1.1 Resource Sharing in Server Processors

Current data centers are equipped with thousands of modern high-performance multi-core processors. Each single processor allows multiple applications to be executed concurrently, increasing resource utilization thanks to resource sharing. As the co-running applications compete for the shared resources and present different resource demands, the performance of each individual application becomes unpredictable. In other words, the inter-application interference affects differently to each application yielding the system to performance unpredictability, which aggravates as the *unfairness* grows. This thesis focuses on analyzing the inter-application interference and proposing management strategies to deal both with performance and unfairness. The research concentrates on two major critical resources: the LLC and the processor cores.

1.1.1. Last Level Cache (LLC)

Modern processors commonly implement a three-level cache hierarchy. The lower cache levels (e.g., L2 and L3) are key components to hide the long main memory latencies, and so they are sized big to reduce the number of accesses to the off-chip main memory. While the L2 cache is private to the core, the L3 or the last level cache (LLC) is shared among all cores. The fact that the LLC is shared allows to improve its utilization and presents important advantages over splitting its storage capacity into smaller private caches [1]. However, sharing the LLC can lead to important shortcomings from a system performance perspective. One of the most harmful effects is known as *cache pollution*, which refers to inaccurate prefetch requests [2, 3] that fill up the cache with blocks that are never (or scarcely) referenced again, replacing other useful blocks.

The destructive interference among applications accessing the LLC increases the number of accesses to the off-chip main memory, which incur long latencies that can severely impact on the system performance. For this reason, the common design choice taken by computer architects is to provide a huge LLC, in the order of a few MBs per core, which adds up to tens of MBs. The capacity becomes even larger when the LLC is built using denser memory technologies, like the eDRAM (embedded DRAM) used in the IBM POWER9 [4] and in the Intel Knights Landing [5] processors.

In recent years, the increasing number of cores and the ongoing pursuit of better performance has motivated the trend towards a higher reliance on private caches (e.g., large L2 caches [6]). Therefore, new cache hierarchy organizations were introduced aimed at keeping a larger amount of private data closer to the processor. Among these designs, non-inclusive caches have been implemented in recent processors [7, 8, 9]. This design choice avoids cache line replication (i.e., in the L2 and the LLC), using the available cache space more efficiently. In this organization, the L2 cache is made larger, and the LLC slice is reduced over that of inclusive LLCs. Overall, the cache space per core is reduced, saving silicon area. The fact that there is less space per core to manage makes cache management more critical in processors with a non-inclusive LLC than with an inclusive LLC. In addition, the number of accesses to the LLC is typically higher since it acts as a victim cache of the L2 cache. This means that a fierce competition for cache space can take place. Consequently, making efficient use of the smaller LLC space is even more critical than in inclusive caches, as the *cache pollution* effect is magnified.

To address the interference at the LLC, some processor manufacturers like Intel, ARM, and AMD have deployed technologies in server processors that allow distributing LLC cache ways among co-running applications. For instance, Intel has deployed *Cache Allocation Technology* (CAT) [10], which allows creating cache partitions and assigning them to applications. The first processors that deployed Intel CAT had an inclusive LLC. This technology has also been included in newer processors with a memory hierarchy containing a non-inclusive LLC.

1.1.2. CPU Cores

Server processors are made up of multiple cores. Most processor manufacturers implement the simultaneous multithreading (SMT) paradigm [11], which allows a physical core to issue instructions from different threads in each cycle. SMT allows increasing the processor throughput; however, instructions from the threads concurrently running on the same core compete, among other shared resources, for the scarce number of issue ports, thus, introducing inter-thread interference that harms their performance with respect to their individual execution. Intel's implementation of the SMT paradigm is known as Hyper-Threading technology [12]. It typically supports the concurrent execution of two threads. From the OS perspective, a given physical core is seen as two logical cores or CPUs. In a multi-core processor with Hyper-Threading cores, it is important to differentiate between physical and logical cores when analyzing resource sharing. Two threads running on different physical cores only share the off-core processor resources, mainly the LLC and main memory. However, when two threads are assigned to the two logical cores of the same physical core, they also compete for intra-core components. These components are critical for performance and include, among others, issue ports, reorder buffer (ROB), physical registers, load queue, store queue, functional units, as well as L1 and L2 caches. As a consequence, the performance of a given thread highly depends on the demands of the internal components of the thread it is co-running with [13].

The current trend in server processors is to increase the number of cores. Recent processors like the 4th generation Xeon Scalable CPUs deploy up to 60 cores [14], and AMD EPYC processors include up to 128 cores [15]. To provide full system utilization, processor cores must be fully subscribed. Many parallel scientific applications [16, 17, 18, 19] provide the capability of launching as many threads/processes as available cores.

Over the last years, research initiatives have been proposed to make parallel applications' number of threads/processes reconfigurable. However, many parallel applications are not capable of scaling as the number of threads increases due to different hardware and software issues: load balancing, data synchronization, cache interference, issue bandwidth, and functional-unit saturation [20, 21, 22]. For this reason, many research efforts have been made to use hardware resources best by adjusting the thread-level parallelism (TLP) of multithreaded applications. More precisely, by assigning to an application the *optimal* number of cores so that it launches the number of threads that achieves maximum performance, avoiding resources being wasted or underused. To this end, parallel applications must be able to adapt the number of threads/processes dynamically at execution time.

1.2 Cloud Computing Paradigm

Cloud computing has evolved as a key computing paradigm. An increasing amount of computing is being performed in public clouds, such as Amazon's EC2 [23], Microsoft Azure [24], and Google Compute Engine [25]. Compared to traditional on-site setups, cloud platforms provide two major advantages for end-users [26, 27]: i) cost efficiency as users can quickly launch jobs without the up-front and operational costs of owning a cluster, and ii) flexibility as users can easily request or release computational resources.

Most public cloud systems deploy powerful computing servers. Following the typical virtualization model, the cloud provider allocates multiple virtual machines (VMs) in the same physical machine, which provides fault isolation, security, and improved manageability. This enables high flexibility and resource control.

1.2.1. Issues and Particularities of Public Cloud Systems

Compared to research carried out in the context of HPC, cloud computing research imposes multiple challenges, especially if research is intended to be applicable to public cloud production systems.

In order to perform representative cloud performance studies, the experimental platform should resemble as much as possible to a real production environment. This fact is especially hard to achieve due to the complexity (both hardware and software) of production systems. Consequently, research works simplify the experimental framework. Unfortunately, existing studies omit important system components [28, 29, 30, 31, 32, 33] or do not consider virtualization with VMs [34, 30, 31, 33], which results in losses of representativeness.

Despite the public cloud can run any kind of workload, an important characteristic that makes public cloud servers different from traditional computing nodes is that they frequently run latency-critical workloads. Most online or interactive services are examples of these workloads. Unlike scientific workloads used in HPC, the performance of latency-critical applications is given by the obtained tail latency, indicated as a percentile (e.g., 95^{th} or 99^{th}) of all the latencies and accounts for the requests that take longer to complete.

A major shortcoming that cloud providers need to face when measuring and evaluating performance is that they do not have information about the applications running on the VMs. This means that VMs are treated as "black boxes".

1.2.2. Resource Sharing and QoS

The fact that cloud providers co-locate multiple VMs in the same physical machine means that inter-VM performance interference will appear due to competition for the major system resources (e.g., processor cores, LLC, or main memory), making performance unpredictable. In other words, the interference adversely impacts on the quality of service (QoS) of the applications. Moreover, when the QoS degrades, it might not comply with the service level objectives (SLOs) stated in the service level agreement (SLA) between a cloud provider and a customer.

To avoid QoS violations and tackle the inter-VM performance interference, cloud providers typically adopt an *overprovisioning* strategy. That is, resources are assigned to each VM in excess to avoid possible performance degradation due to the inter-VM interference. When running latency-critical applications, system resources must be conservatively overprovisioned to ensure compliance with the SLA, as the performance of these workloads is very sensitive to the inter-VM interference. This workaround, however, results in poor utilization of the major resources of the cloud system. For instance, the average CPU utilization is typically far below 50% in cloud servers running latency-critical applications [35] and, in most cases, below 20% [36, 37].

The previous rationale means that an important concern for cloud providers is to reduce the overprovisioning costs of the system resources associated with the VM; in other

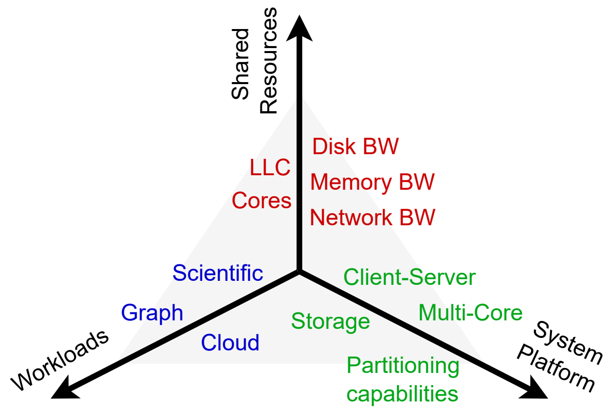


Figure 1.1: The three main axes of research covered in this thesis.

words, there is an interest in improving the resource efficiency to lower the cost of VMs. However, the public cloud imposes significant limitations to implementing a resource management approach. First, it should not have prior knowledge of the workloads running on the VMs (i.e., they should be treated as black boxes). Second, it should not require costly actions (e.g., isolating a VM). And third, it should be general enough to adapt to different workloads and system conditions.

1.3 Objectives of the Thesis

The overall objective of this thesis is to address the inter-application interference at the main system shared resources from two main perspectives: HPC and cloud computing. Within each scope of research, this objective spawns across three main axes. First, the target shared resource(s) to be studied are specified (e.g., the LLC). Then, the workloads to be run are selected (e.g., single-threaded scientific applications), considering the induced interference we intend to study (e.g., cache thrashing behavior). Finally, the system platform is selected depending on the workloads and the specific resource to be studied, and the workload characteristics. Below, we summarize the extent to which this thesis deals with these three axes:

Shared Resources. The main system shared resources addressed in this thesis vary depending on the scope (HPC or cloud) of the study. Regarding HPC, the study focuses on two major processor resources: the LLC and CPU cores. In the context of cloud computing, the main shared resources of the entire system (CPU cores, LLC, main memory bandwidth, disk bandwidth, and network bandwidth) are considered, as public cloud environments cannot obviate any resource.

Workloads. Depending on the scope and the resource to be studied, different benchmark suites are typically used. The manuscript starts by presenting resource management approaches aimed at HPC environments. In this context, we have used scientific single-threaded applications to study the LLC interference and graph parallel workloads to focus the research on core-allocation policies regulating thread-level parallelism (TLP).

However, the popularity and increase in computational load of cloud computing have caused resource sharing to be a concern for cloud providers. Therefore, this thesis evolves the work performed in HPC to cloud computing. In this domain, cloud latency-critical workloads are used to explore resource interference in cloud computing.

System Platform. The experimental setup is highly dependent on both the workload characteristics and target shared resource(s) chosen to be studied. To study resource partitioning in HPC, multi-core processors with resource partitioning tools are required. Two different processor architectures are considered in order to study interference at inclusive and non-inclusive LLCs. In the case of cloud computing, a more complex experimental platform is required with a client-server architecture, a storage node, and a system software stack to implement virtualization. For this purpose, we implement an experimental setup to perform cloud research. Also, the main memory storage capacity and the number of cores (and threads) can be critical elements depending on memory requirements and scalability features of mulithreaded applications.

1.4 Main Contributions of the Thesis

This section summarizes the major contributions of this thesis, grouped according to the two main computing paradigms addressed in this dissertation.

1.4.1. High-Performance Computing (HPC)

The research developed in HPC focuses on the resource management of the LLC (both inclusive and non-inclusive design) and the CPU cores.

- **System TT and throughput improvement via LLC partitioning.** This dissertation proposes LLC partitioning approaches that leverage Intel CAT to improve the performance of multi-program workloads by identifying and protecting the applications whose performance is more damaged by LLC sharing. Newly *problematic* LLC behaviors are identified, which can significantly drop the system performance if not properly dealt with. We also propose an LLC partitioning phase-driven approach that dynamically adjust the partitioning according to changes in the LLC behavior that applications experience during their execution.
- **Containing cache pollution by partitioning of non-inclusive LLC.** Current server processors have redistributed the cache hierarchy space over previous generations, making the LLC smaller but designed as non-inclusive to reduce the number of replicated blocks. Cache management in this organization becomes more critical than in inclusive caches. To address the harmful effect known as LLC pollution, we propose Cache-Poll partitioning policy that, based on the *polluting* behavior and the cache requirements of the co-running applications, distributes the LLC space at run-time among applications. Cache-Poll exploits the non-inclusive LLC design by leaving little room for cache-insensitive and polluting applications.

- **Spatial core-allocation varying TLP transparently.** The default Linux scheduler, which adopts a time-sharing policy to provide a fair scheduler, performs poorly when multiple graph applications share a system. To deal with this issue, we propose AFAIR, a core-allocation policy that dynamically adjusts the number of cores assigned to each graph application based on the shared-memory resource that bottlenecks the processor performance. AFAIR adapts the number of threads spawned by the running applications dynamically without modifying the applications' source code.

1.4.2. Cloud Computing

The work carried out in this thesis on cloud computing is focused on analyzing how the performance of VMs is affected by the consumption of the main system resources considering a public cloud environment. In addition, we outline specific hints cloud providers should take into account to improve the system performance and resource utilization.

- **Experimental platform to carry out controlled cloud research.** To perform cloud research, a small experimental platform is commonly used, which hides the huge system complexity and provides flexibility. Most platforms used in existing work do not include all cloud components or lack the deployment of VMs to provide isolation. To comply with all the main features of cloud production systems, we present Stratus, an experimental platform used to carry out the cloud research presented in this dissertation. Stratus uses VMs to isolate tenant applications, deploys the three types of cloud nodes (server, client, and storage), and manages all main shared system resources (cores, LLC space, memory, network, and disk bandwidth).

- **Characterization of cloud latency-critical workloads.** Understanding how performance is affected by the consumption of the main system resources is a major concern for cloud providers in order to devise virtualization strategies that improve system efficiency. For this aim, a set of representative latency-critical applications are characterized, revealing that the performance of some applications does not scale with the number of threads, and the performance of some others is insensitive to the Hyper-Threading technology. To identify these applications at run-time, the utilization trend of the major system resources is analyzed. In addition to CPU, we have also studied how assigning the share of other major shared system resources to each application impacts on performance.

- **Detecting and estimating QoS degradation in the public cloud.** To prevent QoS violations, cloud providers adopt overprovisioning strategies, but they reduce server utilization and increase operational costs. A mechanism that accurately estimates performance degradation dynamically in a production system would allow cloud providers to improve the servers' utilization. This thesis proposes Cloud White, an approach that is able to detect the inter-VM interference in scenarios with multiple co-located latency-critical VMs and estimate the performance degradation using multi-variable regression models. Unlike previous proposals, Cloud White estimates performance in terms of tail latency and considers the limitations of a public cloud production system.

1.5 Thesis Outline

The remainder of this thesis is organized into eight chapters. Chapter 2 discusses the related work and state-of-the-art approaches. Chapter 3 presents the experimental setup designed to carry out the experiments presented in this thesis. The rest of the chapters are grouped into two main blocks according to the scope of the research.

Regarding the research performed on HPC, Chapters 4 and 5 study the interference at the LLC and propose cache partitioning approaches for inclusive and non-inclusive LLC, respectively. Chapter 6 focuses on cores resource management, presenting a core-allocation policy to improve the co-location performance of graph workloads.

With respect to the work performed on cloud computing, Chapter 7 analyzes the behavior of cloud workloads from a performance and resource-consumption perspective, considering the scalability, load, thread-allocation, and inter-VM interference. Chapter 8 proposes a dynamic approach to detect inter-VM interference and estimate the resulting QoS degradation.

Finally, Chapter 9 summarizes the main contributions of this thesis, discusses future work, and lists the related publications.

CHAPTER 2

State of the Art

This chapter describes relevant previous work on the topics covered in this dissertation. First, the relevant work on resource management is introduced, analyzing approaches that target HPC and Cloud Computing. Second, works analyzing the performance of emerging workloads are revised. Finally, we discuss key state-of-the-art approaches proposed to detect and estimate performance degradation in cloud production systems.

2.1 Resource Management Approaches

2.1.1. Last Level Cache (LLC)

The first approaches dealing with cache partitioning were implemented using simulation frameworks. Some approaches like UCP [38], ASM [39], Vantage [40], PriSM [41] need to modify the eviction and insertion policies to partition the cache; hence they cannot be implemented in existing processors. Other approaches, like the filter cache [42] and ROCA [43], split the cache into different structures to reduce interference.

An alternative approach has been to perform cache partitioning using software tools. Many are based on *page-coloring*, a mechanism that allows controlling which cache sets are assigned to the application data. Among them, an interesting proposal is the work by Liu et al. [44], which proposes a multi-policy memory allocation approach, profiling at run-time the slowdown obtained by applications when the cache space is reduced and selecting the best partitioning policy (horizontal partitioning, vertical partitioning or randomized page interleaving policy) based on this characterization. Park et al. [45] profile applications' cache behavior at page level to assess the page reusability. Then, the LLC is divided into two partitions to place low and high reusability pages together.

Recently, the research trend has changed as some recent processors from distinct vendors provide support to partition the cache [10, 8, 9], so nowadays, the main focus is on implementing cache partitioning policies in commercial processors.

Inclusive Caches

The works focusing on inclusive L3 caches aim to distribute a high number of cache ways, 20 or more in most processors, among the co-running applications. They mainly differ in the type of the applications (e.g., latency-critical, best-effort, or high-priority) co-running in the processor, which receive different treatment according to the aim of the approach (e.g., quality of service of latency-critical, fairness, or turnaround time). These works can be split into two main groups depending on whether they concentrate on HPC or cloud workloads. Below these groups are discussed.

In the context of HPC, Selfa et al. [1] cluster applications using the k-means algorithm and distribute cache ways between the groups, giving exponentially more space to the applications whose performance suffer more from the interference at the LLC to improve system fairness. El-Sayed et al. [46] also group applications into clusters, assigning them to different CLOS. While it significantly improves throughput in selected workloads, it uses detailed profiling, making KPart a more complex solution than those proposed in this thesis. Similarly, DICER [47] uses a similar profiling to perform cache partitioning where high-priority and best-effort applications are executed together. DCAPS [48] proposes a framework based on predictors that use miss rate curves and LLC occupancy predictions. This approach estimates the LLC occupancy from the number of misses incurred, which can lead to wrong conclusions since some applications present a low number of LLC misses but a high number of memory accesses due to prefetches, resulting in a high LLC occupancy. Contrary to this work, we measure the effective LLC occupancy to identify anomalous LLC behaviors that can drop the performance of the co-runners. Sun et al. [49] propose combining cache partitioning with prefetch control to reduce prefetcher-caused inter-core interference. Authors designed the Coordinated Multi-resource Management (CMM) framework to detect prefetch-aggressive applications and allocate resources (prefetcher and cache space) accordingly.

In the context of cloud computing, research works [50, 31, 51] have been published dealing with multiple hardware and software resource management mechanisms (among them, cache partitioning) for cloud systems. These systems present workloads with particular characteristics, like latency-critical applications, where the quality of service (QoS) must be satisfied, which run jointly with best-effort applications to improve resource utilization. Multiple resources are tuned simultaneously (e.g., the number of cache ways and cores) to meet QoS requirements. While Heracles [50] only considers a single latency-critical application placed in a private partition, Parties [31] can manage multiple latency-critical applications. In [51] Funaro et al. use a marked-driven auction system to partition the LLC into isolated partitions. In this way, each guest can bid based on the number of resources it wants to use.

Non-Inclusive Caches

Non-inclusive L3 caches (i.e., LLC) are the newest design trend; therefore, the most recent works have been performed in processors with non-inclusive LLCs.

Some approaches are designed for scientific workloads, targeting system throughput and fairness. POCAT [52] uses *Intel Top-down Microarchitecture Analysis Method* (TMAM), leveraging machine learning to predict applications' IPC for different cache sizes. This model also captures changes in the IPC behavior. However, it requires using a machine learning dataset created by previously collecting TMAM metrics and IPC values of each application for each cache way setting. Xiao et al. [53] identify prefetching sensitive applications to drive cache partitioning using an online profiling phase. Each prefetching-sensitive application is assigned a single private cache way, and prefetching-insensitive applications share the remaining cache ways. Park et al. [54] focus on workload consolidation of commodity servers. Both memory bandwidth and the L3 cache are studied and partitioned together. However, the focus is on improving system fairness rather than system performance. Saez et al. [55] propose LFOC+, a cache-clustering policy that uses dynamic profiling to classify applications. Like [54], LFOC+ targets system fairness. Chatterjee et al. [56] use compiler-generated information to collect information about the memory footprint, cache sensitivity, reuse behavior, and phase timing to estimate the applications' memory requirements. With this information, they propose a proactive cache partitioning scheme that dynamically partitions the cache and schedules processes.

Most current data centers include processors with non-inclusive LLCs; thus, the latest research in resource management in cloud computing has focused on these processors. CLITE [30] propose multi-resource partitioning. It explores a model-learned relationship between job performance and the assigned resource shares (number of cores, LLC ways, memory bandwidth, memory capacity, disk bandwidth, and/or network bandwidth) using Bayesian Optimization. Alita [29] proposes eliminating interference by throttling resource polluters. Regarding the LLC, Alita isolates LLC polluters in one or few cache ways, avoiding cache thrashing and disturbing VM's with *normal* behavior, which share all LLC space. Themis [57] manages memory subsystem resources to provide fair resource sharing while meeting QoS requirements of latency-critical applications. It clusters VMs with similar memory bandwidth consumption to the same CLOS and configures the number of LLC ways and memory bandwidth throttling value accordingly.

2.1.2. Cores

Recent works have proposed core allocation strategies to mitigate resource contention in multi-core processors.

Kundan et al. [58] characterize the pressure that applications inflict on both LLC and memory bandwidth to schedule applications to the processor cores, prioritizing those applications that made the least progress. However, offline information (IPC of the application alone) is required to estimate the progress. In addition, Hyper-Threading is disabled.

Some works have exploited the fact that current parallel application runtimes (e.g., OMP and MPI) support changing the number of spawned threads dynamically to improve resource efficiency and system throughput. Galante et al. [59] have surveyed the approaches proposed in this area in different memory architectures (shared and distributed) and computing environments (cloud and fog). In [60], authors propose a dynamic resizing of malleable parallel applications scheduler for distributed-memory clusters. The aim is to improve cluster utilization and job execution time. However, the code of applications must be changed to make them resizable. Yang et al. [61] present a specific solution for iterative workloads running on Apache Spark named iSpark. Iterative applications usually present initially a high CPU usage, which diminishes as the execution continues. Thus, the authors propose a provisioning approach that dynamically changes the number of allocated long-running processes (e.g., executors) to avoid resource wastage and handle under-utilization. SCALO [62] estimates scalability at runtime considering dynamic contention effects to control the threads allocated to each running OpenMP multi-threaded application. Authors instrument OpenMP runtime to enable setting the degree of parallelism during the execution of parallel regions and allow communication with the SCALO daemon. TBFT [63] combines dynamic thread adjustment on OpenMP applications (without modifying or recompiling the code) with dynamic boosting mode tuning with the objective of optimizing EDP (energy-delay product). NuPoCo [64] aims to maximize system utilization while minimizing execution time by controlling the degree of parallelism of co-located parallel applications and scheduling them in NUMA multi-socket multi-core systems. The authors implement a dynamic loop scheduler in the OpenMP runtime system to provide dynamic spatial scheduling. A similar approach, MAPPER [65], focuses on ensuring QoS rather than maximizing system utilization. MAPPER may run without runtime modification (i.e., MAPPER applies only the scheduling part), but minor performance improvements are obtained.

In the context of scheduling, approaches have been proposed to optimize the execution of concurrent parallel application execution. Some works [22, 66] aim to improve performance by selecting the target number of threads (i.e., thread-level parallelism) each application should execute. Then, given a set of applications, the best scheduling order is decided based on this fact and the available hardware resources. These approaches, however, provide little flexibility as they require knowing the running applications before the execution starts and do not allow adapting the number of spawned threads dynamically. PredG [67] uses machine learning to select the best thread and data mapping policies to run graph applications on a NUMA system. Both approaches require knowing application-level information, such as the input graphs for decision-making.

2.2 Emerging Workloads in Server Processors

2.2.1. Graph Workloads

Much research on graph applications has focused on proposing new hardware [68, 69, 70, 71] or the use of accelerators [72, 73, 74, 75] for optimizing graph processing performance. However, these solutions are not currently available in data centers. Therefore, commodity servers (i.e., general-purpose processors) are the popular choice to execute graph workloads.

Some previous works have characterized the performance of graph applications both under simulators and on real processors to guide future hardware design optimizations for graph processing. Most works [76, 71, 77, 78] focus on the memory subsystem's impact on graph applications. The main conclusions of these works are the following: i) good performance scalability is obtained with increasing numbers of cores [77, 76]; ii) graph applications are memory-latency bound since they do not fully utilize the memory bandwidth [76, 77, 78]; iii) graph applications do show some locality [76, 78] and can benefit from caching, being performance more sensitive to the LLC (L3 cache) space than to the L2 cache [71]; iv) SMT introduces modest performance improvements [78, 76].

Other works focus the study on specific domains, such as the impact of branch prediction on performance [79, 80], hardware prefetchers [81, 76], architectural enhancements, and software optimizations [82] and energy efficiency [83].

2.2.2. Cloud Workloads

Due to the nature and fast evolution of the public cloud and related industry concerns, many research works have been proposed in the last few years. These studies apply a wide range of characterization methodologies that present significant differences regarding virtualization levels (e.g., VMs, containers, or no virtualization), performance metrics (e.g., execution time, throughput, or tail latency), target shared resources (e.g., LLC, main memory, etc.), and system configurations (e.g., SMT vs. no SMT, system specifications). In contrast to previous research works, in this thesis, we perform a comprehensive study including all the main shared resources (i.e., CPU, LLC, main memory, network, and disk); focusing on a realistic configuration for cloud providers with a full virtual machine-oriented system stack and SMT-enabled CPUs; and considering tail latency as a key metric to evaluate QoS from the tenant perspective.

The workload characterization performed in previous works is often leveraged to propose novel resource management approaches dealing with distinct shared resources that affect the behavior of these applications. These approaches significantly differ in whether the target workloads represent public cloud deployments. This is the case when the workload is implemented using virtual machines (VMs) running in full-stack system configurations. This section summarizes previous research taking into account this differentiation.

Approaches Not Considering VMs

In [54], Park et al. focus on the interference at the LLC and/or memory bandwidth, but they do not consider other major shared resources in the cloud environment like the network or the disk. This approach only characterizes each application's behavior according to the number of LLC misses and accesses per second. However, the system performance is not considered at all. Moreover, the approach mainly focuses on HPC workloads, and only a small section studies the behavior of scenarios where a single latency-critical application runs concurrently with other batch workloads.

Also, regarding memory bandwidth, in [84], the impact of Intel Memory Bandwidth Allocation (MBA) technology on performance is studied. MBA provides a relatively coarse interface to limit main memory bandwidth consumption. Therefore, this approach also studies complementary techniques such as thread packing [85] and clock modulation [86].

Jeatsa et al. [87] propose CASY, a cache allocation system for serverless functions in Function as a service (FaaS) platforms. CASY leverages machine learning models to predict the amount of cache to be assigned to each function based on the input data size.

In [88], each application is categorized considering four main aspects that affect performance: scale-up (amount of resources per server), scale-out (number of servers per workload), server configuration, and interference (symbiotic workloads). This categorization can be used to establish the right amount of resources allocated to an application to reach a given performance level while maximizing overall resource utilization (i.e., avoid over-provisioning). The resources considered are the number of compute cores, memory, and storage capacity. Unlike our work, neither LLC occupancy, main memory bandwidth, nor disk bandwidth are considered.

In [50], similarly to our work, Lo et al. characterize the impact of interference at shared resources on performance for different load levels; however, just three latency-critical Google workloads are characterized. To study the effect of interference, synthetic benchmarks stress each shared resource. Nevertheless, the interference at the disk is not analyzed. In contrast, Parties [31] considers disk interference, analyzing six latency-critical applications. However, Parties, as some of the works mentioned above, studies applications running in Linux containers, which are more light-weighted than full VMs. Thus, the studied workloads are not representative of a significant amount of public cloud deployments.

Full System Stack Approaches with VM Support

In [89], a mix of batch (Hadoop running over Mahout and Spark jobs) and latency-critical applications (memcached jobs) are studied in three representative workload scenarios (minimum, medium, and high load variability). This study focuses on finding the optimal mapping of applications, in terms of the number of virtual CPUs, to reserved and on-demand VM instances. Therefore, there is no insight into the exact resources (e.g., cache, disk bandwidth) that affect the jobs' execution time.

To determine the best VM to physical core mapping, in [90], VMs are classified as compute or memory intensive, considering readings from several performance counters. In particular, only three performance events (L2 cache misses, L1 cache misses, and committed instructions) are used to classify applications. The focus is on best-effort scenarios, and the evaluated workloads are four-application mixes composed of SPEC CPU2006 benchmarks, which are scheduled in pairs to the processor cores.

Finally, DeepDive [91] pursues to identify interference between VMs in Infrastructure as a service (IaaS) clouds. Deepdive uses about a dozen low-level metrics (including performance events acting at the L2 cache as the LLC in the system, the iostat and netstat tools, and available hypervisor -VM- statistics) to find out if interference is introduced as well as what is the main shared resource where interference rises. Nevertheless, Deep-Dive lacks important performance events, which are present in modern processors, and partitioning mechanisms (i.e., Intel CAT and MBA) dealing with shared resources like the LLC and the main memory bandwidth. The analysis introduces unexpected metrics, like using the private L1 cache misses, mainly because the L2 is the LLC. In modern processors, the latency of most L1 cache misses is hidden by the out-of-order mechanisms. Experiments use three cloud and latency-critical (web search) workloads.

2.3 Interference in Cloud Systems

2.3.1. Interference Detection Based on Simple Measurements

There is extensive literature on detecting interference due to co-located jobs or resource sharing. Many approaches [92, 31, 93, 88], however, monitor QoS (e.g., in terms of average latency or queries per second), which cloud providers cannot carry out in real production systems since VMs should be handled as black boxes.

Recent approaches [32, 29] have tried to detect inter-VM interference while complying with public cloud limitations. In [32], Javadi et al. propose Scavenger, a resource manager that considers tenant workloads as black boxes and identifies performance interference by monitoring the usage of a subset of the major system resources (memory, network, LLC, and CPU) consumed by the VM. Chen et al. propose Alita [29], which identifies contention online considering a different subset of the system resources (memory bus, LLC, and power supply) based on low-level metrics but omitting important shared resources like disk and network. These approaches present two main shortcomings. On the one hand, the performance interference is detected in a subset of the system resources. On the other hand, no prediction is made on how performance interference impacts on the QoS of VMs.

2.3.2. Performance Interference Prediction Models

Prior works have proposed models to estimate the impact on performance caused by the interference (e.g., introduced by co-runners or limited resources) in distinct environments or from different perspectives. In [94], online prediction models are built from a user-level perspective. Microbenchmarks are run to estimate resource contention at the shared resources, and then application-specific models are used to estimate its impact on performance. Instead, this document focuses on the cloud provider's point of view, and therefore, we assume no knowledge of the application running within each VM.

Another important piece of research [95, 96, 33, 91] has focused on predicting the performance interference that background applications introduce when co-located with latency-critical applications. Other works like [97] focus on the co-location of HPC applications.

All of these works [95, 96, 33, 97] access to data about the performance of individual applications, which is not accessible by cloud providers, making these approaches not practical in real production systems.

DeepDive [91] detects performance interference based on the aggregated resource system utilization of VMs by monitoring low-level metrics. Nevertheless, an important weakness of DeepDive is the *isolation-based methodology* used to quantify interference in VMs, which is prohibitive in public cloud environments. Resource Central [36], based on that certain VMs show consistent behaviors over multiple lifetimes, learns offline from past behaviors to predict the future online behavior of the customer's VM. This approach predicts high-level metrics like CPU utilization, cores, memory, or workload class. To this end, it uses machine-learning methods (e.g., random forest) to output buckets instead of predicting a single number through regression models.

Some approaches [98, 99, 30] use prediction models to estimate the resulting performance when modifying a given resource configuration. Rusty [98] leverages neural networks to make resource and energy consumption predictions, but performance is quantified in terms of IPC and not tail latency. Twig [99] uses performance counters to estimate the QoS that results from different dynamic voltage and frequency scaling (DVFS) and core combinations. CLITE [30] elaborates prediction models to search for near-optimal resource partitioning. However, both Twig and CLITE require monitoring the performance (e.g., latency) of the executing applications at run-time, thus not considering VMs as black boxes. In addition, performance degradation due to the interference is continuously being estimated instead of only when detected.

CHAPTER 3

Experimental Framework

This chapter describes the experimental framework used to conduct the experiments presented in this thesis.

First, the experimental platforms are presented. Two Intel multi-core processors have been used to carry out the work on HPC, each from a different microarchitecture. Regarding Cloud Computing, we have deployed an experimental platform (Stratus) to perform controlled cloud research, focusing on intra-node interference.

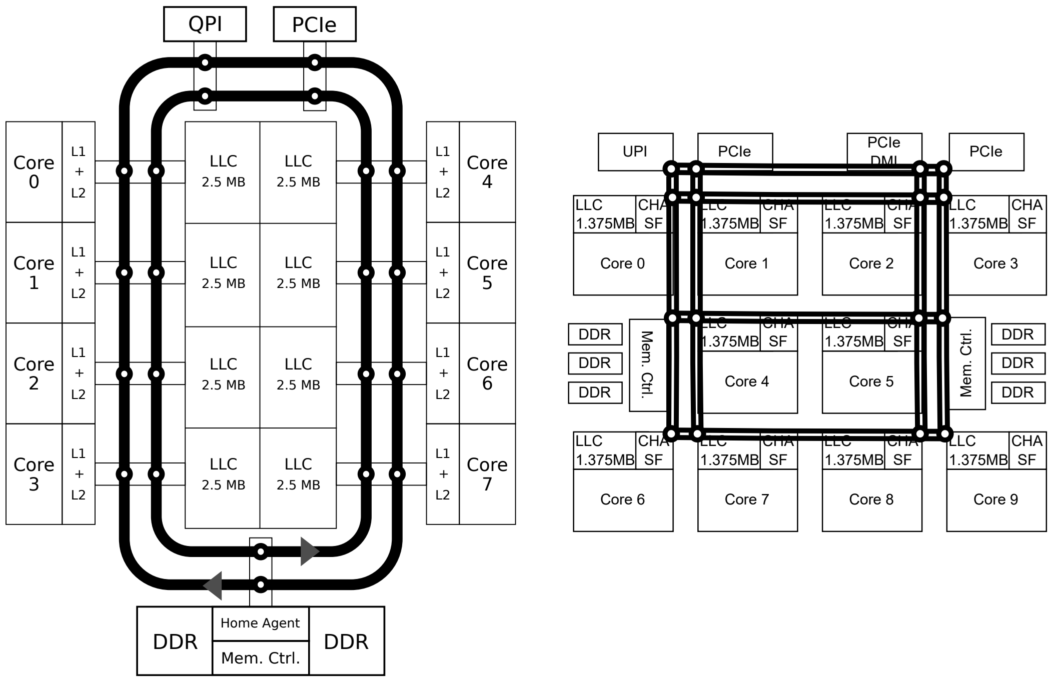
Then, we present the Resource and Application Manager developed to automate the execution of experiments carried out in this thesis, as well as monitor and partition the system-shared resources. This manager has been used both in the work conducted in HPC and Cloud Computing.

Finally, this chapter describes the benchmark suites used.

3.1 Experimental Platforms

Two main Intel multi-core processors have been used to obtain the experimental results of the work performed in this thesis.

- **Intel Broadwell Processor.** The work performed in inclusive LLC (Chapter 4) has been carried out in an Intel Xeon E5-2620 v4 processor, which features eight cores supporting Hyper-Threading. This processor implements a three-level cache hierarchy, with a 20-way 20MB (1MB /way) LLC. Regarding main memory, it has two 16GB memory channels, making a total DRAM capacity of 32GB. It supports the Intel Resource Directory Technologies (RDT) and includes up to sixteen Classes of Service or CLOS (see Section 3.3.1).
- **Intel Skylake-X Processor.** The research carried out in non-inclusive LLC (Chapter 5) and in the server node in the cloud computing part of this thesis (Chapters 7 and 8) has been performed in a 12-core Intel Xeon Silver 4116 processor running at



(a) Broadwell Processor **(b)** Skylake-X Processor

Figure 3.1: Block diagram of the Intel Xeon Broadwell and Skylake processors with low core count (LCC SoC) to illustrate the differences among both microarchitectures.

2.1GHz. Each core has a 1-MB private L2 cache, and all cores share a 16.5MB 11-way (1.5MB/way) non-inclusive L3 cache. This processor supports also supports Intel Resource Directory Technologies (RDT), but the maximum number of supported CLOS is 8. The machine has six main memory channels, holding each 16GB, which amounts to a 96GB (6x16GB) DRAM. It supports a maximum bandwidth of 58 GB/s.

Figure 3.1 shows a block diagram of the memory and the cores of both the Intel Broadwell and Skylake-X procesors. The Skylake-X architecture belongs to the Intel Xeon Scalable processors, which marks a new era in the Xeon processors segments as it includes many enhancements [100, 101].

Improved interconnection network. A mesh interconnect is used instead of a ring bus to interconnect cores, as the latter did not scale well with an increased number of cores.

Redistribution of cache space. Intel Skylake-X provides larger L2 cache space (see Chapter 5). Regarding the LLC, the available space per core is lower and presents a tile design. It minimizes communication traffic as it does not require access to the network on chip (NoC) to access its LLC slice.

Support to more memory channels. Intel Skylake-X servers support up to three channels on each memory controller, making a total of six memory channels.

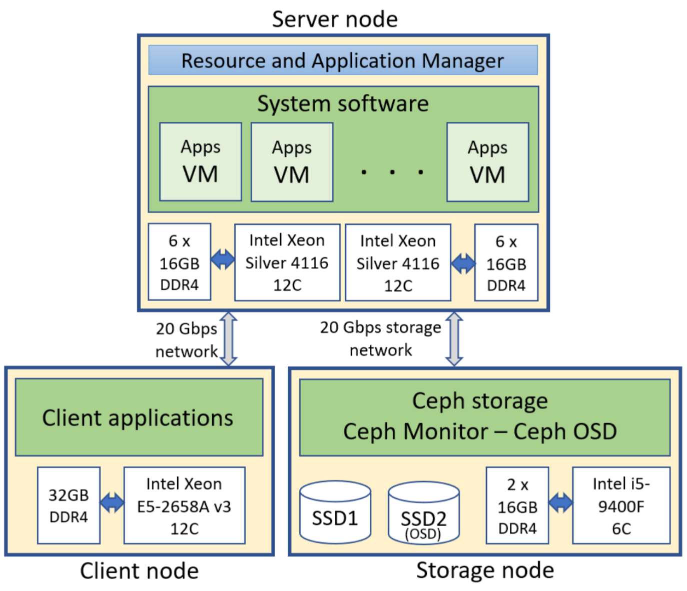


Figure 3.2: Overview of the experimental framework developed for controlled cloud research.

New UPI technology and modular PCIe design The new Intel Ultra Path Interconnect (UPI) replaces the QuickPath Interconnect (QPI) to support higher I/O traffic between nodes. Regarding PCIe, the traffic no longer flows to a single point; up to three separate PCIe lanes are included.

3.2 Cloud Computing Infrastructure

To conduct research in cloud computing, an experimental platform named Stratus has been developed to perform controlled experiments. Stratus has been developed following the guidelines provided by Huawei Cloud since this platform has been built under a research and development agreement with Huawei Technologies Co., Ltd.

Stratus comprises three physical servers (main, client, and storage nodes) interconnected with two 20 Gbps dedicated links. Figure 3.2 shows a block diagram of the experimental platform, including the three system nodes: main node, client node, and storage node, and the installed packages and libraries. Notice that the focus of the research of this thesis is on the interference at the server node (i.e., intra-node interference); therefore, one server node is enough for this purpose. However, Stratus can be easily expanded by adding more server nodes.

The design choices to set up Stratus' experimental platform are taken in two main axes: the deployed hardware and the system software. This section presents and motivates the choices we selected for each axis.

Node	Processor Package		Main Memory
	Processor	#Cores (#Threads)	
Main	2x Intel Xeon Silver 4116	24 (48)	12 x DDR4-2666 16GB DIMMs
Client	Intel E5-2658A	12 (24)	2 x DDR4-2133 16GB DIMMs
Storage	Intel i5-9400F	6 (6)	2 x DDR4-2666 16GB DIMMs

Table 3.1: Node hardware specifications (processor package and main memory).

3.2.1. Deployed Hardware

A key design decision is selecting the node types and the amount of them in the experimental framework. However, the huge complexity of managing a high number of machines that are found in real environments should be avoided. A cloud system includes two main types of nodes: computing nodes and storage nodes. For results to be representative of real scenarios, a minimal experimental framework requires at least one node of each of these types. In addition, a separate node is needed to emulate client behavior.

Table 3.1 shows the specifications (processor package and main memory) for each of the nodes. The specifications of the server and storage nodes can be considered representative of the nodes implemented in real cloud systems. For example, Google Cloud CPU platforms [102], Amazon EC2 C6 and C5 instances [23], and Huawei Elastic Cloud servers [103] use Intel Xeon Scalable Processors including from tens to hundreds of gigabytes of main memory capacity.

Regarding the storage media in the storage node (shown in Figure 3.2), it is composed of two SSD devices. The first one (SSD1) with 500GB of capacity contains the storage node OS and system software, while the second one (SSD2) with 960GB is exclusively devoted to acting as remote persistent storage for server applications running in the server node VMs. This persistent storage is served to the **server** node as a *Ceph Object Storage Device (OSD)*. Ceph [104] is an open-source distributed storage platform that is commonly installed in cloud environments.

Finally, the client and storage nodes are interconnected to the server node with two dedicated 20 Gbps networks. Specifically, the server node has two 20 Gbps network cards (dual port, 10 Gbps per port) that connect to the client and storage nodes.

3.2.2. System Software

To build our experimental framework, we analyzed the main components of OpenStack [105]. OpenStack is an open-source cloud computing software stack, and it is a *de facto* standard for managing virtual services in both public and private clouds. OpenStack is a complex software framework with multiple components and add-ons supporting different types of hardware devices (e.g., network devices, storage devices, etc.) from

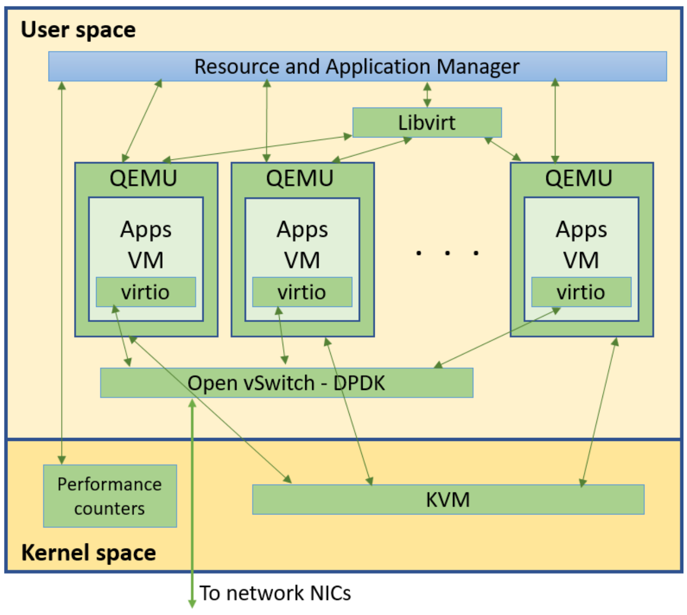


Figure 3.3: Stratus' system software deployed in the server node.

multiple vendors. To avoid dealing with such complexity, we built a simple framework that includes the main software components that can be found in a typical OpenStack deployment. The major simplification lies on the top software levels, which aims to reduce management complexity but has a negligible or null impact on performance monitoring and partitioning, which is the main purpose of Stratus' framework.

Figure 3.3 presents a block diagram of the main components of the system software deployed in the server node and their interactions. These components, which manage the VMs and the network interconnections, are described below.

VM Infrastructure. The execution and management of VMs involves a complex software stack, where three main levels can be distinguished: the hypervisor, the virtual resource manager, and the guest OS and applications. The hypervisor refers to the OS installed in the host physical machine (PM). A wide set of both open-source and proprietary hypervisors are currently being used in the industry. Examples of open-source hypervisors are Linux with KVM [106] and Xen [107]. The former is one of the current industry trends, and it is being used by Amazon [108] and Google [109]. The latter is also supported by Amazon. The virtualization manager refers to the software platform that manages the PM hardware resources and distributes them among VMs. One example of a virtualization manager is Libvirt [110]. Some virtualizers, like QEMU [111], are also supported by KVM and Xen. Finally, the guest OS and tenant applications run in the different VMs. Guest OS and applications can be either proprietary or open source (e.g., a Linux server distribution executing several Internet services).

Network Software. To interconnect the VMs with the physical network interface cards (NICs) of the server node, a *virtual switch* is used. The virtual switch is set up with Open vSwitch (OvS) [112]. Emulated NICs at the VMs (i.e., virtio [113] NICs) and each physical

NIC (both ports) in the server node are accessed from the virtual switch through the Data Plane Development Kit (DPDK) [114]. DPDK enables the direct transfer of packets between virtio NICs and physical NICs, bypassing the host OS kernel network stack. This setup boosts network performance compared to the default packet forwarding mechanism implemented in the Linux kernel.

3.3 Resource and Application Manager

Providing an experimental framework that allows automating the setup and execution of experiments plays a key role in speeding up the research experiments.

For this aim, we have developed a resource and application manager to assist the researcher, providing a user-friendly interface. It implements three main functions: i) managing and controlling the execution of one or more applications, whether it is executing with a VM or not, ii) monitoring hardware performance counters and system resource utilization, iii) helping partition system resources and assigning them to the applications or VMs.

Below, first, we explain how each of the system's shared resources is monitored and partitioned. Then, to illustrate the functioning of the developed manager, we explain the execution of experiments when launching experiments with VMs.

3.3.1. Monitoring & Partitioning Main Shared Resources

Applications executing together in the same processor compete for the system's shared resources. This means that the performance of a given application (or VM) will depend on the co-running applications. In other words, on the share of the resource that the application is able to use. As a consequence, it is worth studying to which extent the performance of a given application is affected by varying the amount of share allocated to the application.

Server processors provide support to monitoring events via hardware performance counters. These events count the occurrence of microarchitectural events, such as the number of retired instructions or misses at a specific cache memory level. Our Resource and Application Manager uses **Linux Perf** [115] to configure and access performance counters.

In addition, in the last few years, server processors have been provided with advanced technologies that allow monitoring and partitioning the major system resources. Below, we explain how the monitoring and partitioning of each shared resource is implemented in the Resource and Application Manager without relying on any external tool.

CPU cores. The processors used to perform the experiments implement Intel Hyper-Threading technology [12], that is, Intel's implementation of the simultaneous multi-threading (SMT) paradigm [11]. Hyper-Threading processors typically support the concurrent execution of two threads. From the operating system perspective, a given physical core is seen as two logical cores or CPUs. Thus, applications threads or processes can

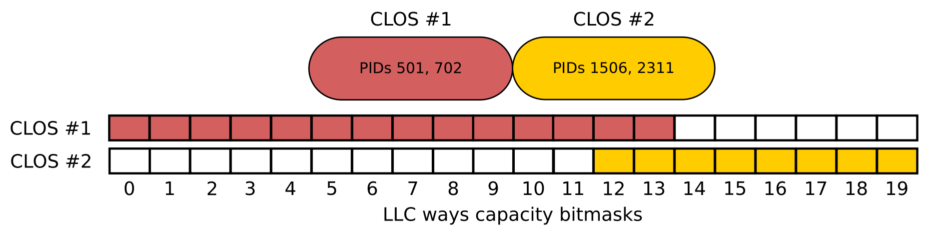


Figure 3.4: Intel CAT example with PIDs associated to two CLOSes.

be allocated or pinned to different logical cores using Linux CPU affinity system calls[1]. One of the most popular metrics regarding CPU monitoring is CPU utilization, which accounts for the percentage of time a CPU is active. It is a crucial metric in cloud environments since CPU utilization has been proven to be low (less than 20%) most of the time [35, 37]. Thus, many resource provisioning strategies [36, 32, 116] seek to improve the CPU usage. To obtain the utilization of each CPU, we use the data collected from the file `/proc/stat`, which reports statistics about the kernel activity aggregated since the system first booted. To pin the VMs' virtual CPUs (vCPUs) to logical cores of the physical machine, Stratus uses Libvirt's API [110].

Last Level Cache (LLC). Due to the high latency to access to main memory upon LLC misses, the LLC is one of the *key* shared resources in current multi-core processors. Recently, some processor manufacturers like Intel have developed technologies that allow monitoring and partitioning of the LLC. In Intel processors, these technologies are known as Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT) [10]. The LLC is partitioned in a *per way* basis; that is, a cache way acts as the granularity size allocated to Classes of Service (CLOS). A CLOS can be defined either as groups of applications (PIDs) or as groups of logical cores to which a partition of the LLC is assigned. For each CLOS, the user has to specify i) the subset of ways that can be written and ii) which applications or logical cores belong to the CLOS. The cache ways that can be written by the applications belonging to a CLOS are defined with a *capacity bitmask* (CBM). Cache ways are not necessarily private to a CLOS, as they can be shared with other CLOSes by overlapping the CBMs. Figure 3.4 shows an example of a possible CAT configuration using 2 CLOSes and a shared cache with 20 ways. However, this technology is constrained so the ways assigned to a given CLOS have to be consecutive.

Memory Bandwidth. Memory bandwidth can considerably impact the performance or responsiveness of applications. For instance, in a server system with different VMs accessing the main memory, the inter-VM interference can significantly grow and make the most memory-sensitive VMs perform below an acceptable level, compromising the QoS. Recent Intel Xeon Scalable processors introduce Memory Bandwidth Allocation (MBA) [117], which allows to distribute memory bandwidth between the running applications. More precisely, it allows controlling the memory bandwidth between the L2 and the L3 (i.e., LLC) caches. Similarly to CAT, MBA works using CLOS. That is, MBA bandwidth limits apply only to CLOS, to which the user can assign tasks (PIDs) or cores. However, MBA **works on a per-core basis**. If the individual memory bandwidths of two

[1]`https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html`

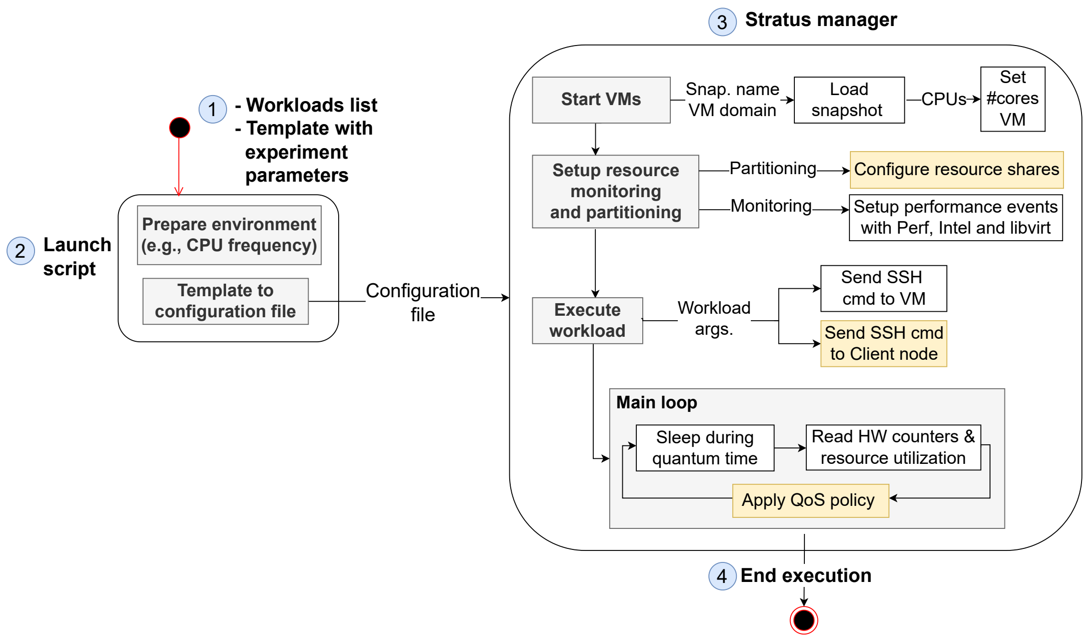

3 Stratus manager
1 - Workloads list
- Template with experiment parameters
2 Launch script
Prepare environment (e.g., CPU frequency)
Template to configuration file
Configuration file
Start VMs
Snap. name VM domain
Load snapshot
CPUs
Set #cores VM
Setup resource monitoring and partitioning
Partitioning
Configure resource shares
Monitoring
Setup performance events with Perf, Intel and libvirt
Execute workload
Workload args.
Send SSH cmd to VM
Send SSH cmd to Client node
Main loop
Sleep during quantum time
Read HW counters & resource utilization
Apply QoS policy
4 End execution

Figure 3.5: Workflow followed by the Resource and Application Manager to launch experiments when running one or more VMs. Yellow boxes represent optional actions.

applications running on the same core are limited with different values, the *maximum* limitation is the one that will apply to that core.

Disk Bandwidth. Many workloads that operate on big data files or databases exceed the main memory size. Consequently, these workloads must constantly rely on the I/O system to access the disk and load/store the required data. Monitoring and partitioning this subsystem is, therefore, of high interest. I/O access to the disks can be monitored using the virsh tool or Libvirt's API. Both mechanisms offer the same functionality and allow monitoring the number of read, write, and flush operations, the number of bytes read and written, as well as the total duration of the read, write, and flush operations.

Network Bandwidth. VMs running on the same physical machine share network resources whose bandwidth and latency play an important role in the QoS of tenant applications. Consequently, network resources should be monitored and partitioned to minimize the inter-VM interference. The number of network packets or bytes that go through a network interface can be monitored with Libvirt's API.

3.3.2. Manager in Action: Execution of Experiments with VMs

To illustrate how the developed Resource and Application Manager performs experiments, Figure 3.5 identifies the main steps performed when launching one or more VMs together with the applications to be run on them (*VM-application pairs*). Next, each of the steps is discussed in detail.

Define experiment workload and parameters. As a prior step, the workload (i.e., VMs and applications to be run on them) and experimental conditions must be defined. To ease this task, Stratus makes use of MAKO templates [118], which provide a simple and intuitive language to specify the parameters of the experiments: VMs and applications to be executed (domain name, workload, number of CPUs, etc.), vCPUs core pinning, performance events to be monitored, length of the quantum, etc. The template can also specify if VMs are only allowed to use a partition of a shared resource (LLC, memory bandwidth, network bandwidth, or disk bandwidth).

Execute *Launch* script to start the manager. To start running an experiment, the user executes the *launch* script. First, the script prepares the execution environment. For instance, fixing the processor frequency to avoid variability among experiments. Additionally, the processor clocks of both the server and client machines are synchronized to ensure server- and client-collected metrics are aligned, using the Network Time Protocol (NTP) [119] with a known NTP time server (europe.pool.ntp.org). When the environment is ready, the configuration file is generated with the workloads to execute and all the experiment's parameters from the MAKO template. Then, the manager starts to run.

Prepare VMs for execution. First, the manager performs sets up and starts the VMs. To reduce the start-up overhead, the manager makes use of the *snapshots* feature of Libvirt. A snapshot is a copy of the state of a VM, including the disk and main memory contents. This feature preserves a VM's actual state and data at a given time. Therefore, this state can be reverted at any moment. For each VM, we have taken a snapshot that has already performed the OS boot process and is ready to receive the command to launch the target benchmark. Once the VMs are started and the snapshots are loaded, the number of CPUs of each VM (i.e., vCPUs) can be modified in case a multi-threaded application is going to be executed and more than one CPU is required.

Setup resource monitoring and partitioning. With QEMU, each vCPU is associated with a processor ID (PID) in the host OS. These PIDs are required to monitor hardware performance counters with Perf individually for each thread (i.e., vCPU) of the VM. Similarly, LLC and memory bandwidth monitoring is performed on a PID basis. The remaining resources, network, and disk bandwidth are monitored per VM. The manager also allows the partitioning of the main system shared resources and assigns each VM a share of a given resource. Therefore, a resource share is allocated to the VM if specified in the template.

Start running applications in the VMs. When the VMs are operative and ready to start executing the applications, an SSH command is sent to each VM to start the execution of each workload. Stratus' manager is adapted to support the execution of client-server workloads (e.g., TailBench benchmark suite) as well as best-effort or batch workloads (e.g., stressor microbenchmarks or SPEC CPU benchmarks). In case of a client-server workload, an SSH command is sent to the client node to start running the clients, which send requests to the server (already running).

Perform actions in each quantum. Once the execution starts, the manager executes the *main loop* (see Figure 3.5) for the rest of the execution time. In each iteration, the manager is suspended for a given quantum length (established in the template). Then, data are collected from different sources (e.g., hardware performance counters, Linux file system, Intel library, or Libvirt) to monitor the main system resources (CPU usage, LLC occupancy, memory, network, and disk bandwidth). Additionally, the manager is adapted to allow implementing and applying QoS policies. For instance, policies that manage resource sharing among VMs [31, 32, 29], predict interference among VMs [33, 30, 120, 121] or schedule VMs [122].

Execution end. The main loop ends when the manager detects that all the VMs have finished running their applications. At that time, it shuts down the running VMs.

All the data collected from the hardware performance counters and system resources are stored in CSV files, ready to be processed. Additionally, for characterization and debugging purposes, statistics and data are also collected inside the VMs. For instance, in Tailbench workloads, the clients report results such as latency per query, queries per interval, tail latency, etc.

3.4 Benchmarks

To quantify the interference in current multi-core processors within HPC and Cloud Computing, popular and standard benchmark suites have been used, covering a wide range of domains and applications. In addition, distinct *microbenchmarks* have been used to evaluate specific components (see Chapter 8). Below, each of the studied benchmark suites is presented.

3.4.1. Scientific Computing: SPEC CPU2006 and 2017

To evaluate the proposal to LLC partitioning proposals presented in Chapters 4 and 5, we have used single-threaded applications from the SPEC CPU 2006 [123] and SPEC CPU 2017 [124] suites. These benchmark suites are made up of 50 applications and *kernels*, representative of scientific applications of common use in the scientific community, used to evaluate the processor performance.

For the SPEC CPU2006 benchmarks, we used the input files *ref* or reference to perform experiments that focus on performance measurement.

Regarding SPEC CPU2017, the SPECrate suite has been used. These benchmarks aim to run multiple concurrent copies of each benchmark and measure throughput. Like SPEC CPU2006, both SPEC integer and SPEC floating-point benchmarks were used.

3.4.2. Graph Computing: GAP

To evaluate graph processing, have used graph applications from the GAP Benchmark Suite [18]. This benchmark suite includes a set of graph algorithms (kernels) representative of a wide set of application domains (e.g., social networks, science) and five synthetic and real-world input graphs. We chose GAP since it provides the reference implementation of graph algorithms without the need to execute them under a framework, thus avoiding framework-related overheads.

In Chapter 6, we evaluate the algorithms Betweenness Centrality (`bc`), Breadth-First Search (`bfs`), Connected Components (`cc`), PageRank (`pr`), and Single-Source Shortest Path (`sssp`); and the graphs Kron, Twitter, Urand, and Web. The combination of an algorithm with an input graph will be referred to as a graph application (e.g., *prKron*, where pr is the PageRank graph algorithm, and Kron is the input graph). Graph algorithms from GAP are parallelized using OpenMP.

3.4.3. Cloud Benchmarks: Tailbench and CloudSuite

Latency-critical applications are increasingly common in data centers. These applications typically support online interactive services (e.g., web search) and must respond to the input requests within certain latency bounds to guarantee QoS (e.g., the 95^{th} or 99^{th} percentile latency) and provide a satisfactory user experience.

As latency-critical applications, we use the TailBench benchmark suite [125]. This suite includes eight representative workloads of today's latency-critical applications. Below, the main characteristics of the studied applications are described:

- `img-dnn` is a handwriting recognition application based on OpenCV. The application uses randomly chosen samples from the MNIST database.
- `masstree` is a fast in-memory key-value store written in C++. Each user's request often involves many tens or hundreds of requests to the key-value store; therefore, it has very short latency requirements.
- `moses` is a statistical machine translation system written in C++. It is driven using randomly-chosen dialogue snippets from the opensubtitles.org English-Spanish corpus.
- `shore` is a transactional on-disk database. It uses the industry-standard OLTP benchmark TPC-C. Its database and logs are both stored in a solid-state drive.
- `silo` is an in-memory transactional database designed for modern multicores. It uses TPC-C like *shore*, although they differ significantly in how they store and access data.
- `specjbb` is an industry-standard Java middleware benchmark. Java middleware is widely used in business services and must often satisfy strict latency constraints.
- `sphinx` is a compute-intensive speech recognition system written in C++. Speech recognition systems are an important component of speech-based interfaces and applications such as Apple Siri, Google Now, and IBM Speech to Text.
- `xapian` is a search engine written in C++ that is widely used both in software frameworks (e.g., Catalyst) and popular websites (e.g., the Debian wiki).

Tailbench client requests are issued to the server following the Zipfian distribution, which accurately models request times in online services [126] [127]. We extended the Tailbench source code to report the tail latency of the requests serviced dynamically after a configurable time slice, set to 1 second in our experiments.

Cloud providers often need to evaluate the efficiency of servers that stream multimedia content. However, none of the studied Tailbench applications exhibit such behavior since they all present negligible network demands. Because of this reason, other applications outside the Tailbench benchmark suite need to be considered. With this aim, this work also analyzes the `media-streaming` workload from CloudSuite [19]. This application, based on the NGINX web server, is a streaming server that hosts synthetic videos of various lengths and qualities. The server is accessed by clients based on the *httperf's wsesslog* session generator, which performs a set of requests per session for videos stored in the server. This popular application in today's data centers makes it possible to broaden the spectrum of studied behaviors further.

3.5 Metrics

A wide set of metrics have been used to evaluate the performance of the research proposals presented in this dissertation. Below these metrics are presented, classified into two main groups according to the granularity, application, or system they evaluate.

3.5.1. Application-Level Metrics

For the work performed in HPC, application-level metrics quantify the individual application's performance considering both the overall system as well as its specific performance on the major system resources:

- **Slowdown.** This metric defines the performance loss an application experiences considering the entire system, in terms of execution time, with respect to a *baseline* system.
- **IPC.** It is the number of instructions committed per processor cycle. This is the universal metric commonly used to quantify the overall system performance of single-threaded applications.
- **Individual components.** Specific metrics have been evaluated for each individual component. For instance, regarding the LLC, the following metrics can be used:
  - **HPKI_LLC.** It refers to the number of hits in the LLC every thousand committed instructions.
  - **MPKI_LLC.** It refers to the number of misses in the LLC every thousand committed instructions.

  - **MPKC_LLC.** It refers to the number of misses in the LLC every thousand clock cycles. This metric is useful when comparing the access rate of different applications.
  - **Occupancy.** LLC space, quantified in MB, used by an application.

In the context of cloud computing, metrics cover both the performance and resource utilization of the running VMs. As in the presented experiments a VM only hosts an individual (single- or multi-threaded) application, we classify VM performance metrics in application-level metrics.

- **Tail Latency.** It quantifies the response time of the requests that take longer to complete than those falling in a given percentile (e.g., 95^{th}).
- **Response Time.** It defines the average time the server takes to reply to client requests.
- **CPU utilization.** This metric refers to the average CPU utilization of the logical core(s) being evaluated.
- **LLC occupancy and main memory bandwidth.** Sum of the total LLC space in (MB) occupied and main memory bandwidth (MB/s or GB/s) consumed, respectively, by the vCPU cores.
- **Disk and Network bandwidths.** They refer to the bandwidth consumed, in MB/s, at the disk and network, respectively, by a given VM.

3.5.2. System-Level Metrics

These metrics are used when multiple applications are running in the system.

- **Turnaround time (TT).** It is the time elapsed between the start and the end of the execution of a program. In this thesis, it refers to the time that elapses from the start of the execution of a workload mix to the moment at which the last application of the mix ends. TT is considered one of the primary performance criteria in general-purpose systems and interactive environments [128] as it is related to resource utilization.
- **Average Normalized Turnaround Time (ANTT).** For a given workload mix, this metric is calculated as the arithmetic mean of the slowdown of each application that makes up the mix. It is a complementary metric to be studied alongside the TT [129], as it quantifies the performance losses of individual applications.
- **Fairness and Unfairness.** Because of the inter-application interference at the shared system resources, the performance of some applications can suffer severely. This metric estimates how fairly and equitably the system resources are distributed among co-running applications. In this thesis, both fairness and unfairness metrics are used, quantified using the coefficient of variation [130]:

$$Unfairness = \frac{\sigma_{slowdown}}{\mu_{slowdown}} \tag{3.1}$$

$$Fairness = 1 - Unfairness \tag{3.2}$$

- **Mean IPC.** It is defined as the average IPC across all the applications that make up the workload mix. This dissertation uses the geometric mean of IPC since the arithmetic mean or raw-IPC can yield misleading conclusions [131]. This metric is used to quantify system throughput.

Part I

High-Performance Computing (HPC)

CHAPTER 4

Inclusive LLC Resource Management

Applications executing in modern high-performance multi-core processors compete for shared resources. Among these resources, the Last Level Cache (LLC), typically shared among all the cores, plays a key role in the final performance.

Cache sharing yields the system to important issues from a performance perspective. These problems appear due to the inter-application interference at the shared cache, making the system performance become unpredictable. Some processor manufacturers like Intel, ARM, and AMD have recently deployed technologies that allow distributing LLC cache ways among co-running applications. For instance, Intel has deployed *Cache Allocation Technology* (CAT), which is being delivered in recent server processors.

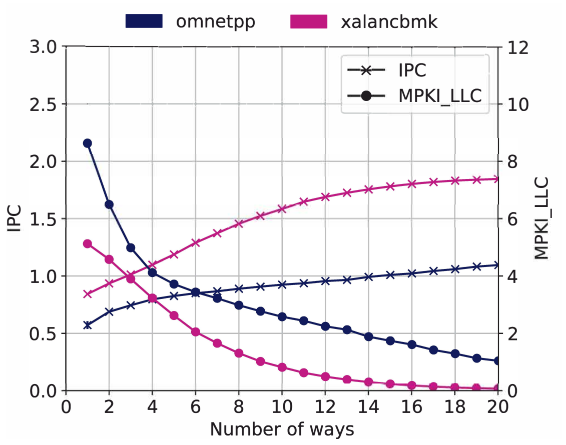
Several cache partitioning approaches leveraging Intel CAT have been proposed addressing different performance targets like system fairness [1] or system throughput [46]. However, these works present several drawbacks:

1. **Complex solutions.** They require modifying the kernel [48], offline information [49] or performing a profiling phase [46, 47] where applications are executed with different cache sizes, incurring a high overhead.

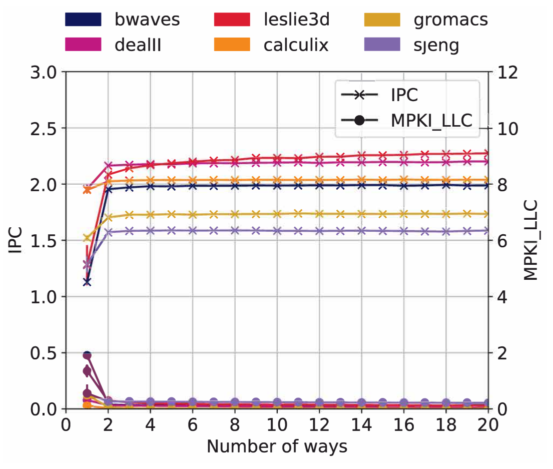
2. **Harm the performance of individual applications.** They benefit some specific applications (e.g., to improve fairness [1]) at the expense of damaging the best-performing ones, which has a negative impact on the overall system throughput.

3. **Unseen behaviors.** Do not address important cache behaviors present in current workloads, such as low cache reuse [1].

This chapter presents the two LLC partitioning approaches proposed in this dissertation, Critical-Aware (CA) and Critical-Phase Aware (CPA), that leverage Intel *Cache Allocation Technologies* (CAT) to partition the cache and assign subsets of cache ways to groups of applications based on the dynamic performance at run-time. Both CA and CPA manage to reduce the turnaround time (TT) while sustaining (and even improving) the system performance in terms of IPC by forcing applications to use the LLC space effectively.



(a) Critical applications



(b) Non-critical applications

Figure 4.1: Example of the average behavior of representative critical and non-critical applications when varying the LLC space.

To design the partitioning algorithms, we performed a detailed characterization of the behavior of SPEC CPU 2006 and 2017 benchmark suites. The study identifies the different behaviors applications experience from the LLC perspective.

4.1 Application Sensitivity to the LLC Space

To characterize the behavior of SPEC CPU2006 and SPEC CPU2017 applications[1] from the LLC perspective, which illustrates how the amount of cache space affects the overall performance, we executed each application varying the assigned cache space from 1 to 20 ways. The study was carried out in an Intel machine with a 20-way 20MB (1MB/way) LLC with Intel CAT support (see Chapter 3 for further details).

4.1.1. Critical and Non-Critical Behaviors

After analyzing the results, we concluded that the applications could be divided in terms of how the cache space influences performance, at first glance, into two main categories:

- **Critical applications**. In these applications, an increase in the assigned LLC space results in increased performance, quantified with the instructions per cycle (IPC). Similarly, with a limited number of cache ways, they present a high number of cache misses in the LLC (misses per kilo instruction or MPKI_LLC).
- **Non-critical applications**. These applications exhibit a contrasting behavior since their performance does not improve by increasing the amount of LLC space over

[1]Since there are applications whose name appears in both suites, from now on, the suffix _06 and _17 will be added to specify the corresponding suite.

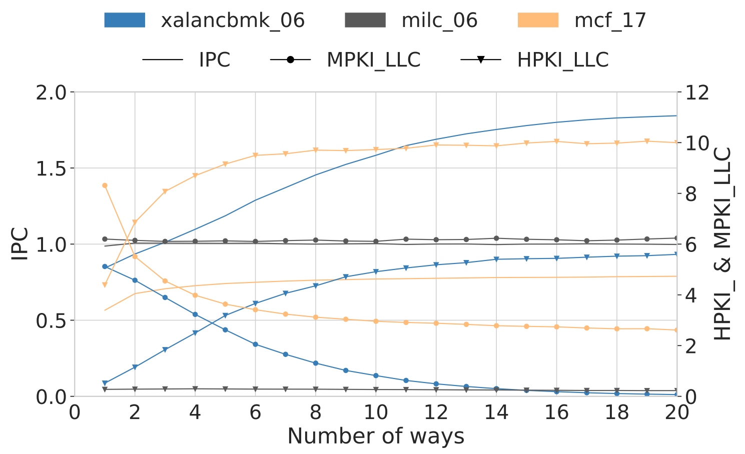


Figure 4.2: Average behavior of critical and problematic applications varying the LLC space.

two ways[2]. That is, the average IPC and MPKI_LLC remain constant when varying the assigned number of ways. Additionally, the MPKI_LLC is particularly low, which allows to easily distinguish this type of application from the critical one at run-time.

Figure 4.1 illustrates the behaviors explained above with two plots, one representing two critical applications, `xalancbmk` and `omnetpp`, and another with six non-critical applications, `bwaves`, `dealII`, `leslie3d`, `calculix`, `gromacs`, and `sjeng`, all from the SPEC CPU 2006 benchmark suite [123]. Note that both plots have two Y-axis scales, one for the IPC (left Y-axis, data points represented with crosses) and another for the MPKI_LLC (right Y-axis, data points represented with dots). The X-axis depicts the number of cache ways a given application was allowed to use when executed alone in our experimental platform, so for each application and cache space, we know the average IPC and MPKI_LLC.

4.1.2. Problematic Behaviors

A broader and deeper evaluation of an extensive range of workload mixes revealed that some applications do not fit well in any of the two categories (i.e., critical and non-critical). We found that, even though some applications have a high MPKI_LLC (a characteristic previously identified as being of critical applications), their performance did not improve with larger cache partitions. In contrast, the overall system performance was harmed, as these applications increased the inter-application interference. Applications with these behaviors will be referred to as **problematic**.

Looking further in this direction, we found that cache block *reuse* can assist in distinguishing critical and problematic behaviors. The reuse can be quantified with the hits per kilo-instruction (HPKI_LLC) metric. Figure 4.2 shows the IPC, MPKI_LLC and HPKI_LLC average values of three applications (`xalancbmk_06`, `milc_06`, and `mcf_17`), all of them identified as critical. For each application, the values shown in the graph were obtained

[2]With one way, the LLC behaves as a direct-mapped cache, and performance drops considerably.

in isolated execution, increasing the number of assigned LLC ways from 1 to 20 (the entire cache). As it can be observed, all of them present an average MPKI_LLC higher than 2 (an average value expected in critical applications). With the exception of `xalancbmk_06`, the IPC does not significantly increase beyond two ways. For instance, `milc_06`'s IPC slightly improves initially with two cache ways and then remains constant. Similarly, the IPC improvement of `mcf_17` is scarce compared to that obtained by `xalancbmk_06` with larger partitions. Therefore, `milc_06` and `mcf_17` can be classified as problematic applications.

Looking at the HPKI_LLC in Figure 4.2, we can make three interesting observations. Firstly, `xalancbmk_06`'s HPKI_LLC grows with the number of LLC ways, and, conversely, the MPKI_LLC decreases. As in other critical applications, this means that the performance (IPC) achieved by `xalancbmk_06` improves with the amount of LLC space. Secondly, in contrast, `milc_06`'s HPKI_LLC is always almost constant and close to 0, regardless of the number of LLC ways that it has been assigned. This is because LLC blocks are scarcely reused (accessed again) or not reused at all before being evicted. Consequently, the performance improvements are negligible with additional cache space. In this work, we refer to this kind of problematic behavior as **squanderer**. Other contemporary works [46, 38, 50, 132] also identify this type of cache polluter behavior; however, the work proposed in this dissertation treats applications with this behavior differently.

Finally, `mcf_17` presents the highest HPKI_LLC, which increases when the application is granted additional LLC space. Nevertheless, assigning more cache ways over a given number does not improve its performance. The reason is that the out-of-order execution is not able to hide the latency of most accesses to the LLC, which eventually causes stalls and prevents further performance gains. Moreover, there is an important number of LLC misses, even with large LLC partitions, probably due to `mcf_17`'s cache access patterns being difficult to predict and prefetch effectively. This kind of problematic behavior will be referred to as **bully** and, to the best of our knowledge, it has not been identified in any previous work.

4.1.3. Medium Behavior

Taking a closer look at the behavior of critical applications as initially identified, we found that some of them presented a more relaxed behavior; that is, they showed a higher IPC and lower MPKI_LLC than other more critical applications. That is, different degrees of *criticality* can be appreciated. Figure 4.3 illustrates this fact by plotting the average IPC and MPKI_LLC of two critical applications (`xalancbmk_06` and `blender_17`), and, for comparison purposes, a non-critical application (`gromacs_06`). As in Figure 4.2, the values were obtained in isolated execution, increasing the number of assigned LLC ways.

In contrast to `gromacs_06`, whose performance is not affected, both critical applications present a significant performance degradation with just two cache ways (2 MB). Nevertheless, compared to `xalancbmk_06`, which experiences a poor IPC (less than 1) with this small cache space, `blender_17` presents a mild IPC (around 1.4). In addition, `blender_17`'s IPC stabilizes much earlier (with 6 LLC ways) compared to `xalancbmk_06`. Therefore, we

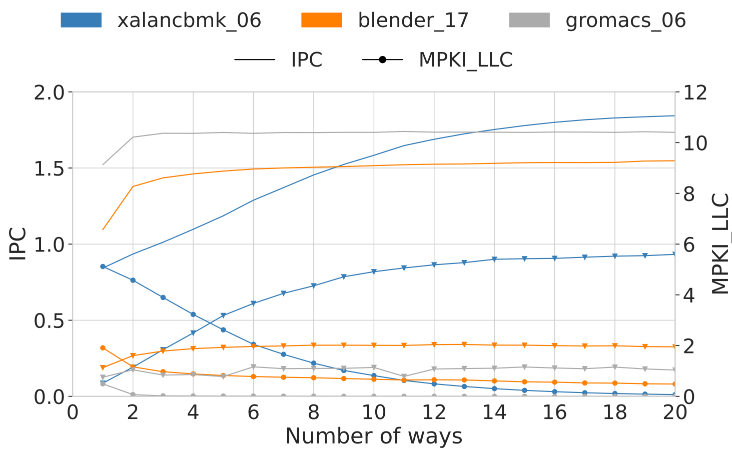


Figure 4.3: Average behavior of critical and non-critical applications varying the LLC space.

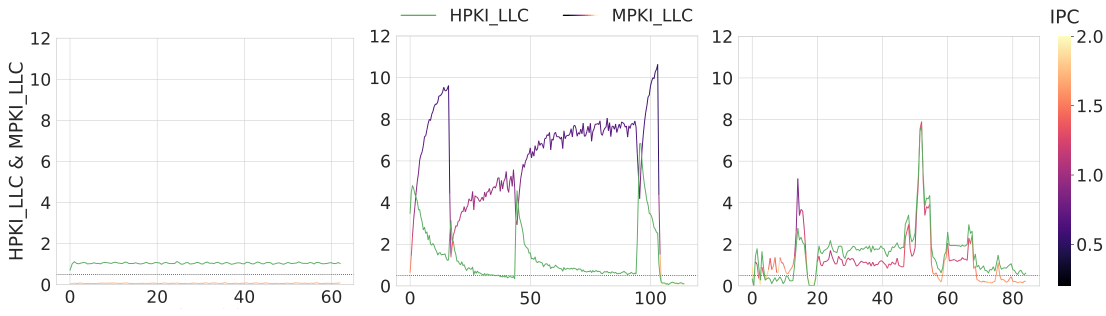
claim that `blender_17` is less critical because: i) it presents an IPC higher than other critical applications with reduced LLC space, and ii) it does not require as much LLC space to maximize its performance. This means that it can be assigned to a smaller LLC partition with a minor impact on its performance. This way, more LLC space could be devoted to improve the performance of other applications. Consequently, CPA must ensure that less critical applications do not occupy more LLC space than needed for performance by monitoring their *LLC occupancy* (see Section 4.2.3 for further details). This less critical behavior will be referred to as **medium**, while the more critical behavior has been named **sensitive**.

4.2 Dynamic Behavior of Applications

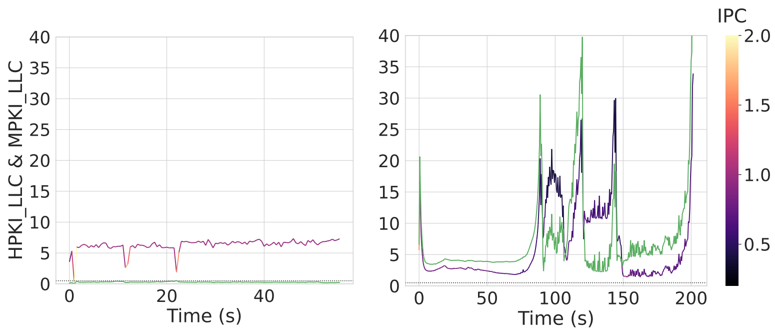
4.2.1. Reaction to Constrained LLC Space

This section shows how applications may present different execution phases by executing them in isolation with two cache ways, the minimum LLC space assigned to an application in this work (1 cache-way partition is not considered as it makes the cache behave as a direct-mapped cache, resulting in poor performance). This analysis assists us in observing how applications behave in each execution phase with limited (and available for itself on average) LLC space.

Figure 4.4 illustrates the dynamic behavior of five applications, each representing one of the previously identified LLC behaviors. The MPKI_LLC metric is depicted with a color-changing line and the HPKI_LLC with a green line. The column on the right side labeled as IPC shows the colormap associated with the IPC values. We found that an MPKI_LLC and HPKI_LLC value below 0.5 means that the LLC does not adversely affect the IPC and that there is negligible data reuse, respectively. Thus, a horizontal dotted black line at $Y = 0.5$ is plotted in each graph to facilitate the analysis. The three upper plots of Figure 4.4 correspond to `gromacs_06`, `xalancbmk_06` and `blender_17`, which show repre-



(a) Non-critical app. `gromacs_06` **(b)** Sensitive app. `xalancbmk_06` **(c)** Medium app. `blender_17`



(d) Squanderer app. `milc_06` **(e)** Bully app. `mcf_17`

Figure 4.4: Dynamic behavior of the different application behaviors with 2 cache ways.

sentative non-critical, sensitive and medium behaviors, respectively. The leftmost graph corresponds to a typical non-critical behavior. `Gromacs_06` presents an MPKI_LLC close to 0 throughout the execution, which yields an IPC relatively high (around 2), despite the limited cache space. The next two graphs illustrate sensitive and medium behaviors, respectively. As observed, `xalancbmk_06`'s IPC decreases as the MPKI_LLC increases; that is, a rise in the MPKI_LLC line matches a color change to a darker color. Across all its execution, this sensitive application presents a relatively low IPC (below 1) and an MPKI_LLC as high as 11.

In contrast, `blender_17`, the medium application, presents a higher IPC than `xalancbmk_06`. Following the same trend, `blender_17` has lower MPKI_LLC than `xalancbmk_06`, although it can be considered high if compared with the MPKI_LLC of non-critical applications, which is close to 0. Thus, we can argue that with a reduced cache space, the impact on performance is much lower for a medium application than for a sensitive one. On the other hand, the HPKI_LLC metric is used to distinguish a critical (sensitive or medium) from a squanderer behavior. This is done by checking that there is at least some LLC reuse; that is, the HPKI_LLC is higher than 0.5 (dotted line at y=0.5).

The two lower plots of Figure 4.4 depict the two different behaviors shown by problematic applications. The left-side graph (Figure 4.4d) presents the behavior of `milc_06`, a squanderer application with low LLC reuse. As observed, it has a high MPKI_LLC (similar to the sensitive application `xalancbmk_06`), but its HPKI_LLC is always lower than

Critical Applications	
Sensitive	**Medium**
gcc_17, omnetpp_06, omnetpp_17, soplex_06, xalancbmk_06, xalancbmk_17	blender_17, cactuBSSN_17, fotonik3d_17, GemsFDTD_06, parest_17, roms_17, sphinx3_06, zeusmp_06

Problematic Applications	
Squanderer	**Bully**
milc_06	mcf_06, mcf_17

Non-critical Applications
astar_06, bwaves_06, bwaves_17, bzip2_06, cactusADM_06, calculix_06, cam4_17, dealII_06, deepsjeng_17, exchange2_17, gamess_06, gobmk_06, gromacs_06, h264ref_06, hmmer_06, imagick_17, leslie3d_06, lbm_06, lbm_17, leela_17, libquantum_06, nab_17, namd_06, namd_17, povray_06, povray_17, perlbench_06, perlbench_17, sjeng_06, tonto_06, wrf_06, wrf_17, x264_17

Table 4.1: Categorization of SPEC CPU 2006 (_06) and SPEC CPU 2017 (_17) applications.

0.5. This behavior is homogeneous throughout the execution. The right-side graph (Figure 4.4e) illustrates the behavior of `mcf_17`, an application that presents a bully behavior. Notice that this application does not present a bully behavior during the whole execution but only in several execution *phases*. For instance, it can be observed that from approximately second 85 to second 150, both the MPKI_LLC and HPKI_LLC are dramatically high (both are above 10 most of the time), and thus, the IPC value drops down to about 0.5. In this type of phase, the performance of `mcf_17` would not significantly improve if more cache ways were assigned to it. This is due to the high amount of time taken by the LLC accesses, as explained in Section 4.1.2. Therefore, even though the amount of LLC hits is very high, this bully application will inevitably achieve poor performance.

4.2.2. Detecting Phase Changes Dynamically

As observed in the previous section, the behavior of phases from a performance (i.e., IPC) point of view can widely differ. We found that *IPC phases* are linked to different LLC space needs or *LLC phases*. For instance, notice that `mcf_17's` behavior, studied above as an example of a bully application, does not show this behavior during the entire execution. In the execution phase that extends from second 5 to second 75 approximately, `mcf_17` behaves as sensitive. In this phase, `mcf_17` has lower HPKI_LLC and MPKI_LLC values than in the plotted bully phase. Similarly, a non-critical behavior appears at the end of the execution of both `xalancbmk_06` and `blender_17`. Table 4.1 classifies the studied applications according to their dominant behavior.

Techniques dealing with phase-change detection have been widely studied in the past [133] [134] [135]. The main goal of detecting a new phase is to select the architectural configuration leading to optimal performance for the incoming phase. In particular, in

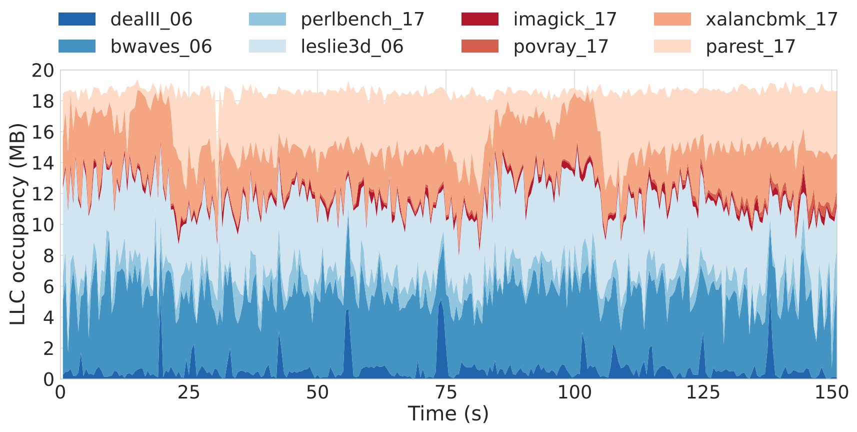


Figure 4.5: Dynamic LLC occupancy of mix #17 with NP.

this work, we use phase-change detection to select the best cache partitioning scheme. We have devised an approach to detect phase changes based on the method proposed by Liao et al. [136], *Interval Coefficient of Variation (ICOV)*, as part of an online phase detection scheme that guides dynamic L2 cache partitioning, implemented using page coloring. ICOV was chosen for two main reasons, simplicity, and effectiveness, as well as due to hardware constraints since only four hardware performance counters are available in our processor. This method measures the homogeneity of a given sample of numbers (IPC values in this work). The ICOV value is calculated for intervals belonging to the same phase according to Equation 1, where *i*, *j*, and *j-1* are the initial interval of the phase, the current one, and the previous one, respectively; and $\overline{IPC_{x,y}}$ refers to the average IPC from interval x and to interval y.

$$ICOV = \frac{\mid IPC_j - \overline{IPC_{i,j-1}} \mid}{\overline{IPC_{i,j}}} \qquad (Eq.1)$$

The lower the ICOV value, the closer the IPC of the current interval is to the phase trend. We found 20% as the best-performing threshold for our proposal and platform.

4.2.3. Multiprogram Execution: Fighting for LLC Space

The previous analysis has shown that applications have different cache occupancy requirements depending on their shown behavior. Non-critical applications require almost no occupancy and present low interference, so they can be placed together in a single partition. Critical applications have significant LLC space requirements, but it is important to distinguish sensitive from medium since the latter does not need so much space. Squanderer applications have little reuse, and therefore, they hardly require any space like non-critical applications. Finally, bully applications need a minimum amount of LLC space in order to not degrade, even more, their performance.

Thus, the aim of the following study is to analyze if the Linux default scheduler (i.e., when no partitioning (NP) policy) assigns applications cache space according to their needs. Figure 4.5 shows an example of the LLC space occupied by each application of one of the studied mixes (#17) under NP. This mix contains one medium application (`parest_17`), one sensitive application (`xalancbmk_17`), and six non-critical applications. Two observations can be made. Firstly, there are non-critical applications like `bwaves_06` and `leslie3d _06` that occupy more space than the critical ones. Secondly, `parest_17` occupies the same (or even more) space than `xalancbmk_17`, despite `xalancbmk_17` having higher space requirements. Notice that CPA correctly addresses both observations by properly identifying the different cache behaviors.

From this example, we can conclude that applications cannot be characterized by only monitoring the LLC occupancy but other LLC-related metrics, like IPC, MPKI, and HPKI, should be monitored instead. Nevertheless, we can still leverage the LLC occupancy of individual applications to check if, once classified, they are using efficiently the assigned cache space. For instance, in the previous example, the LLC occupancy of a non-critical application like `bwaves_06` could be monitored to check if it is using an excessive amount of cache space. Likewise, by monitoring LLC occupancy and performance (IPC) of critical applications, we can detect a wasteful usage of LLC space performed by some medium applications, which do not need as much LLC space as sensitive applications. Further details can be found in Section 4.4.3.

4.3 Critical-Aware Approach

This section summarizes the main design goals of the Critical-Aware (CA) Partitioning Approach. The general idea is to improve system throughput by only using two cache partitions or two CLOS: one for the critical applications and another for the non-critical, assigning a larger amount of LLC space to the critical applications. Reducing the effective space non-critical applications prevents them from damaging the performance of critical applications.

CA consists of three main phases: i) application classification, ii) base partition settings, and iii) dynamic adjusting of partitions, discussed next.

4.3.1. Application Classification

At the beginning of the execution, before carrying out any action, some time is taken to warm up the cache. After that, the algorithm enters the *reset state*, in which the MPKI_LLC of all the applications is computed. The algorithm computes the rolling mean (μ) and standard deviation (σ) for the last 10 intervals of the MPKI_LLC, considering all the applications to detect outliers (that is, critical behavior) by using Miller's criterion [137]. Applications outliers are detected by comparing the MPKI_LLC of the current interval to the result of Equation 4.1.

$$Limit\_outlier\_MPKI\_LLC = \mu + 3 \times \sigma. \tag{4.1}$$

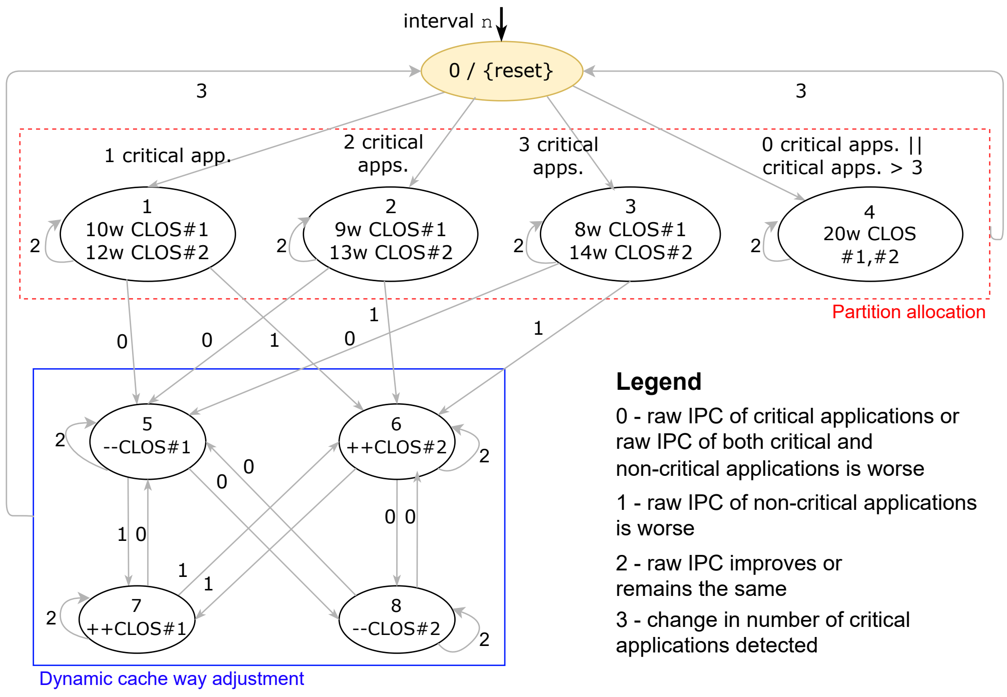

interval n
0 / {reset}
3
3
1 critical app.
2 critical apps.
3 critical apps.
0 critical apps. || critical apps. > 3
1 10w CLOS#1 12w CLOS#2
2 9w CLOS#1 13w CLOS#2
3 8w CLOS#1 14w CLOS#2
4 20w CLOS #1,#2
Partition allocation
5 --CLOS#1
6 ++CLOS#2
7 ++CLOS#1
8 --CLOS#2
Dynamic cache way adjustment
Legend
0 - raw IPC of critical applications or raw IPC of both critical and non-critical applications is worse
1 - raw IPC of non-critical applications is worse
2 - raw IPC improves or remains the same
3 - change in number of critical applications detected

Figure 4.6: State diagram of cache partitioning performed by CA. CLOS #1 contains non-critical applications, and CLOS #2 contains critical applications.

4.3.2. Base Partition Settings

Once applications are classified, the algorithm creates two partitions (CLOS #1 for non-critical and #2 for critical applications), whose sizes depend on the number of critical applications detected. Figure 4.6 shows the state diagram of the cache partitioning performed by CA. This first stage is marked in red.

For instance, in case one critical application is detected, 12 ways are assigned to CLOS #1, and 10 ways are assigned to CLOS #1. The higher the number of critical applications, the more cache ways are assigned to CLOS #2, except when no critical applications are found. Similarly, when there is a majority of critical applications, the cache is configured as a single partition (default configuration).

The initial partitions' layouts were empirically determined based on a deep and thorough study of static configurations, evaluating application mixes with different numbers of critical applications. Notice that a partition size represents a given percentage of the total LLC ways. For instance, with one critical application, 60% of the ways are allocated to the critical application's CLOS and 50% to the non-critical applications' CLOS, having 10% of the cache ways shared between both CLOS. Thus, by applying this observation, the proposed approach could be easily generalized and adapted to be used in another CAT machine deploying an LLC with different associativity.

When the number of critical applications varies due to a change in the behavior of an application (from critical to non-critical or vice-versa), the actual partitioning scheme must be updated. To this end, the algorithm transitions back to the reset state, setting the default cache configuration (all applications in CLOS #1 with 20 ways). Then, the classification process starts again.

4.3.3. Dynamic Adjusting of Partitions

Although the initial partition layouts obtain good performance for a wide set of application mixes, there is not an optimal cache configuration that perfectly suits all of them. For this purpose, CA dynamically adjusts the base configuration at run-time, measuring the effect of such changes in the system performance and using them to guide changes in the base partitioning (*Dynamic cache way adjustment* box in Figure 4.6).

To check the impact of a change in the actual partitioning, CA estimates what would have been the performance if such a change was not made. A simple but effective approach is to assume that the IPC in the next measured interval remains similar to the previous one. This approach provides estimates with around 4% *mean square error* for the studied set of applications.

An issue CA faces when dynamically adjusting partitions is that some applications tend to have phases in which they are critical and phases where they are not. This is challenging because CA resets the partitioning when a change in the number of critical applications occurs. A high number of resets can reduce the potential system throughput. CPA is a phase-change-driven approach, so it deals with this drawback.

4.4 Critical Phase-Aware Proposal

4.4.1. General Overview of the Approach

CPA, like CA, devotes the first intervals of execution to warming up the cache. Initially, all applications are assumed to be non-critical and allocated in CLOS #1, which spans the whole cache (i.e., the default CAT configuration).

Algorithm 1 depicts the pseudo-code of CPA partitioning policy which is applied (after the warm-up phase) periodically in each time interval of the execution. Firstly, in **step 1**, the hardware performance counters are read, and the collected data are used to calculate the inputs to the algorithm. Five main hardware events are sampled for each studied application: instructions (I), cycles (C), and three LLC events ($\#misses, \#hits$, and $occupancy$). The gathered values are used to compute the algorithm's inputs (MPKI_LLC, HPKI_LLC, IPC, and LLC occupancy).

In **step 2**, the ICOV value of each application is computed to detect phase changes, which are detected when the ICOV surpasses the $ICOV_{threshold}$ value. If a phase change is detected, it is checked if a change in the application behavior has occurred. This is done

Algorithm 1 CPA pseudo-code

```
 1: ————— STEP 1 —————
 2: for all apps do
 3:     Read I, C, and LLC events (#misses, #hits, occupancy) and compute metrics: $IPC, MPKI, HPKI$
 4: end for
 5: ————— STEP 2 —————
 6: if First Interval then
 7:     $update\_clos = true$
 8: else
 9:     for all applications do
10:         Compute $ICOV$
11:         if $ICOV > ICOV_{threshold}$ & application behavior change then
12:             $update\_clos = true$
13:         end if
14:     end for
15: end if
16: if $update\_clos$ then
17:     Update cache settings according to Table 3; return
18: end if
19: ————— STEP 3 —————
20: for all critical applications do
21:     if $IPC > th_{IPC,L}$ & $LLCoccup > LLCoccup_{critical}/2$ then
22:         Reduce the #ways assigned to the CLOS
23:     end if
24: end for
25: if $NumCritical == 1$ & $NumMedium == 1$ then
26:     Enlarge CLOS #1 to leverage the free space
27: end if
28: ————— STEP 4 —————
29: for all non-critical apps do
30:     if $LLCoccup > LLC_{CLOS1}/3$ & $MPKI < 0.5$ & $HPKI < 0.5$ then
31:         Isolate application in a CLOS with few ways
32:     end if
33: end for
34: ————— STEP 5 —————
35: if $NumCritical \geq 1$ then
36:     Adjust cache sizes of CLOS #1 and critical CLOS(es) like CA
37: end if
```

by comparing the inputs of the algorithm with the thresholds presented in Table 5.2. For instance, one application is categorized as bully if its IPC is very low (less than $th_{IPC,VL}$), and both its MPKI_LLC and HPKI_LLC are very high (greater than $th_{MPKI,VH}$ and $th_{HPKI,VH}$, respectively). If a change in the application behavior is found, the cache configuration is updated according Table 4.3. This configuration update may imply moving the application from one CLOS to another (e.g., from CLOS #1 to CLOS #2) and/or updating the bit masks or the number of cache ways assigned to one or more CLOS.

The next steps further refine the cache configuration by determining if the applications assigned to the cache partitions are *behaving properly* regarding their LLC occupancy. That is, CPA checks if an application is using more cache space than it needs. In **step 3**, critical applications are checked for a medium behavior. Since these applications do not need so much LLC space as sensitive applications (see Section 4.1.3), CPA checks if there is a crit-

Application Category	IPC	LLC MPKI	LLC HPKI
Sensitive	L (< 1.3)	H ($Eq.$4.2)	not VL (≥ 0.5)
Medium	M (≥ 1.3)	H ($Eq.$4.2)	not VL (≥ 0.5)
Bully	VL (≤ 0.6)	VH (≥ 10)	VH (≥ 10)
Squanderer	$\times$	H ($Eq.$4.2)	VL (< 0.5)
Non-critical	*otherwise*		

Table 4.2: Thresholds for each metric and level ($th_{Metric,Level}$) used to identify the categories. Columns define the metric and H (high), M (medium), L (low), VH (very high) and VL (very low) define the levels.

ical application that shows a medium behavior (IPC is higher than $th_{IPC,L}$) and occupies too much cache space (more than half of the cache space occupied by the critical applications $LLCoccup_{critical}$). In such a case, the number of ways assigned to the CLOS holding the medium application is reduced to the proportional part. That is, half if there are one or two critical applications and one-third in case there are three critical applications. Given that each critical application resides individually in one CLOS (see Section 4.4.3 for further details), the space assigned to each critical application can be easily managed. In case there is only one critical application and it is detected as medium, the number of ways of CLOS #1 is increased, so no cache ways are left unused. Note that, in case there is more than one critical application, at least one should be considered as sensitive since halving the space to, for instance, two critical applications will leave too little space for them, damaging their performance.

In **step 4**, non-critical applications are checked. In this case, CPA isolates applications that occupy an excessive amount of LLC space (quantified as more than one-third of CLOS #1's space) and make no profit from it. That is, they show very low reuse (HPKI_LLC) and misses per kilo-instruction (MPKI_LLC). These applications are isolated in a separate CLOS with few cache ways shared with CLOS #1. This cache arrangement prevents these cache-greedy applications from occupying too much cache space.

Finally, in **step 5**, the partition sizes are adjusted as done in CA. This mechanism has been implemented so that it does not let critical applications take too much space and confine the remaining applications to a marginal space. Remark that every time the cache configuration is updated in this step, CPA waits for some idle intervals where it is not adjusted again, leaving some time for applications to take advantage of the additional space or reduce the amount they are using to match the new configuration.

4.4.2. Identifying LLC Behaviors at Run-Time

LLC behaviors identified in previous sections are checked as follows. First, the algorithm checks for a bully behavior. As discussed above, applications presenting this behavior exhibit very high MPKI_LLC and HPKI_LLC values (higher than $th_{MPKI,VH}$ and $th_{HPKI,VH}$, respectively) and very low IPC (less than $th_{IPC,VL}$). Since the performance of these applications does not improve by assigning them a higher amount of cache ways, two so-

CA cache configurations (# of used ways)		
# Critical Apps.	**CLOS #1 ways (mask)**	**CLOS #2 ways (mask)**
1	10 (`0x003ff`)	12 (`0xfff00`)
2	9 (`0x001ff`)	13 (`0xfff80`)
3	8 (`0x000ff`)	14 (`0xfffc0`)
0 or more than 3	20 (`0xfffff`)	20 (`0xfffff`)

CPA extensions		
App. Type	**# CLOS(es)**	**# of Ways (mask)**
Bully/Non-critical	1	Same as CA
Critical	2, 3 or 4	Same as CA
Squanderer	5 or 6	2 ways/CLOS shared with CLOS # 1 (`0x00003` with 1 CLOS, `0x0000f` with 2 CLOS)

Table 4.3: Initial cache mask configurations for CA and extended configurations used in CPA.

lutions could, in principle, apply, i) isolate it in a single and small CLOS or ii) allow it to remain in CLOS #1 together with non-critical applications. We evaluated both design choices and found that the second choice provides the best results because they span a higher number of cache ways.

Second, CPA checks for critical (sensitive and medium) behaviors. Notice that both sensitive and medium behaviors are identified as critical in step 2, but they are differently addressed in step 3, where the LLC space is adjusted accordingly. Critical applications present a high MPKI_LLC (greater than $th_{MPKI,H}$). Note that in Table 5.2 threshold $th_{MPKI,H}$ is not defined by a fixed value but by Equation 4.2. Therefore, this threshold varies depending on the benchmarks that make up the mix, but according to the equation, the threshold value will always be in the *high* level range (i.e., > 1). Although many statistical studies use $3 \times$ std over the μ, we found empirically that a more relaxed threshold (1.5 standard deviations) works well across the studied mixes. Additionally, the equation excludes the MPKI_LLC of applications showing a previously detected problematic behavior from the calculation of the mean and standard deviation since such high values skew the data model. Note too that CPA does not only consider the MPKI_LLC to detect a critical behavior but also takes into account the achieved IPC and the LLC reuse (i.e., HPKI_LLC) (see Section 4.1).

$$th_{MPKI\_LLC,H} = max(1,\ \mu + 1.5 \times std) \tag{4.2}$$

Third, CPA checks for squanderer behavior. An application exhibits this behavior whenever it fulfills two conditions: i) it occupies a significant fraction of the LLC, and ii) it presents low data reuse. The former means that the application experiences a high MPKI_LLC (fulfills Equation 4.2), and the latter that it has a very low HPKI_LLC (less than $th_{HPKI,VL}$). Taking into account the previous rationale, CPA isolates squanderer applications into a separate CLOS with few cache ways (shared with CLOS #1) since no performance benefits are achieved with additional space.

Finally, if the behavior of a given application does not fulfill the criteria of any of the three mentioned behaviors, then it is considered non-critical.

4.4.3. Dynamic CLOS Management

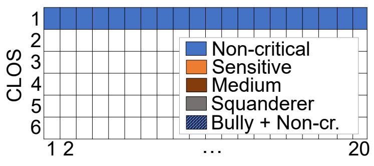
CPA leverages the data collected from the *LLC occupancy* hardware counter, which measures the cache space occupied by each application. To the best of our knowledge, this is the first time this metric has been used to drive a partitioning strategy.

When applications are placed together in a CLOS, the space available is often not shared evenly. Applications present different access rates, and the LLC replacement algorithm is driven, among others, by the access rate. This means that if two applications sharing the same CLOS have widely different access rates, the application accessing with higher frequency will likely occupy much more space than the other. Having an unconstrained number of CLOS allows CPA to use private CLOS to host individual applications and easily control the cache ways assigned to them. Note that a private CLOS does not necessarily means private space since the ways assigned to a CLOS may be shared with another CLOS.

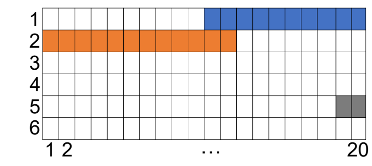
Private CLOS in CPA serve three main purposes, i) limit the interference between medium and sensitive applications, ii) avoid unfair space distribution among non-critical applications and iii) isolate squanderer applications. The first purpose is achieved by placing each critical application in a specific CLOS. The rationale behind this design choice is to facilitate reducing the LLC space to medium applications since they need less space than sensitive applications, so reducing the inter-critical application interference. The second aim refers to non-critical applications. Even though these applications have little space requirements, some non-critical applications occupy more cache space than they need, i.e., the same performance is achieved with less occupancy. This may affect co-runners' performance if they are left with too little space (e.g., less than 1 MB). Thus, these applications are isolated in a separate CLOS with a few cache ways if this situation is detected. Finally, the third aim is achieved by isolating squanderer applications individually in private CLOS with few LLC ways since these applications have little reuse and barely need LLC space. Notice that the space assigned to these CLOS is shared with CLOS #1, unlike previous works which isolated this cache pollutant applications in a private CLOS with private ways (not share ways with other co-runners). This fact, however, reduces the effective cache space that can be accessed by the remaining applications, which is is not the best design choice for performance [1]. Therefore, unlike previous works, we allow the ways assigned to squanderer applications to overlap with other applications. In particular, with the ways assigned to non-critical applications and other problematic applications, which are less affected by LLC interference than critical applications.

4.4.4. Working Example

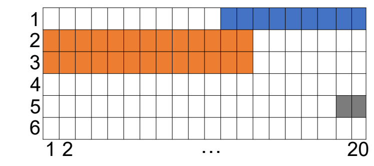
To help understand how CPA works and illustrate how cache partitions are disposed, this section presents a working example of a hypothetical execution scenario considering



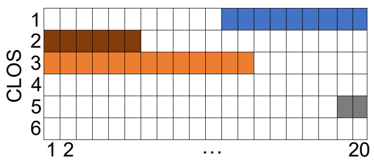
(a) Initial configuration. All applications assigned to CLOS #1.



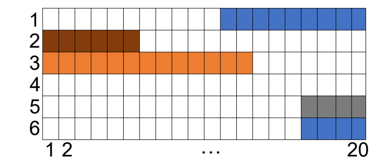
(b) Squanderer (CLOS #4) and critical (CLOS #2) behaviors are detected.



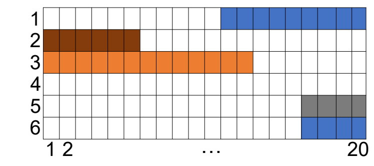
(c) Another application from CLOS #1 is detected as critical (CLOS #3).



(d) Application in CLOS #2 detected as medium.



(e) Non-critical application is isolated in CLOS #5.



(f) Application in CLOS #3 changes its behavior to bully.

Figure 4.7: Cache partitioning example in CPA. Each column represents one cache way, and each row one CLOS.

a mix made up of eight applications. The example studies six different events that occur along the execution. Figure 4.7 shows the active CLOS (following the criteria shown in Table 4.3) and the LLC ways (from 1 to 20 ways) associated with each CLOS on each event.

At the start of the execution (Figure 4.7a) all 8 applications of the workload mix are in CLOS #1, which spans all the cache (default cache configuration). After warming up the cache (at *First Interval*), each application's behavior is checked. Suppose that one of the 8 applications exhibits a critical behavior and another a squanderer. Each application is assigned to a separate CLOS, i.e., two new cache partitions sized with the initial settings are created (see Figure 4.7b). In this case, CLOS #2 holds the critical application and is assigned 12 cache ways (ways #1 to #12, see Table 4.3), and the squanderer application is allocated in CLOS #5 with 2 cache ways (ways #19 and #20). The partition sizes are then dynamically adjusted depending on their LLC requirements. For simplicity, the dynamic adjustment is not shown in this example since it follows a complex state machine. Some cache ways are shared (i.e., overlapped) among CLOS #1 and #2 to improve the cache efficiency.

In the next event, Figure 4.7c, a non-critical application in CLOS #1 experiences a phase change and starts showing a critical behavior. Then, CPA creates a new partition (associated with CLOS #3) to host this application, and the CLOS mask is updated. Notice that each application showing a critical behavior is placed on a different private CLOS, but initially, both partitions share the same ways. Lets assume now that the critical application in CLOS #2 shows a higher IPC than $th_{IPC,L}$ and it occupies a high fraction of the critical LLC space. This application is labeled as medium, and the CLOS #2 size is reduced to half (Figure 4.7d). The next event (Figure 4.7e) assumes that an application in CLOS #1 starts showing a squanderer behavior (wasting the cache space by occupying a high fraction of the partition), but it presents a very low MPKI and data reuse. To counteract this situation, CPA creates a new partition (associated with CLOS #6) to isolate this application. Notice that CLOS #5 and #6 are assigned 4 shared ways instead of 2 cache ways each in order to improve cache efficiency.

Finally, the critical (i.e., sensitive) behavior of the application in CLOS #3 moves to a bully behavior (Figure 4.7f). When this happens, two main actions are carried out. Firstly, the application in CLOS #3 is returned back to CLOS #1. Secondly, CLOS #2 is given back the space it had before halving it, and in forthcoming intervals, this application will be rechecked for a medium behavior.

4.5 Experimental Setup

The experiments have been conducted in the Intel Broadwell processor described in Section 3.1 with the software manager we developed, described in Section 3.3.

4.5.1. Workload Mixes

The workload mixes evaluated in this work were randomly generated using 50 applications: 28 applications from the SPEC CPU 2006 benchmark suite and 22 applications from the SPEC CPU 2017 suite. Table 4.1 shows how applications classify into the defined categories. It can be appreciated that non-critical applications dominate both benchmark suites. Taking this observation into account, 31 mixes consisting of 8 applications each (i.e., the number of cores in the system) were randomly generated, keeping non-critical applications as the dominant group and varying the number of critical and problematic applications from 1 to 3.

Mixes have been ordered according to the number of applications that the mix contains belonging to the critical or problematic categories; that is, the higher the number of the mix, the higher the number of applications from these categories. Mixes #1 to #12 contain one critical or problematic application, mixes #13 to #24 contain two, and mixes #25 to #31 contain three.

4.5.2. Experimental Parameters

In this work, thresholds (upper and/or lower) were empirically determined through thousands of experiments for three main metrics: IPC, MPKI_LLC, and HPKI_LLC. For each metric, different levels have been defined, referred to as Very High (VH), High (H), Medium (M), Low (L), and Very Low (VL). Table 4.2 summarizes the values of the thresholds used to perform the experiments presented in Section 4.6. From now on, we will use the term $th_{Metric,Level}$ to refer to the threshold of a given level for a given metric. Notice that threshold $th_{IPC,L}$ behaves as an upper threshold for medium applications and as a lower threshold for sensitive applications. All metrics have a fixed numeric threshold except for MPKI_LLC, which is determined by Equation 4.2 (Section 4.4.2), a variation of Equation 4.1 (Section 4.3.1) which was used in CA.

In addition to the thresholds in Table 5.2, we set the $ICOV_{threshold}$ to 0.20, the number of *idle* intervals to 5, and the number of *warm-up* intervals to 10. Experiments use CLOS #1 to host non-critical applications, and a new CLOS is allocated whenever a new behavior that requires a private partition is detected. For the 8-application mixes used in this work, CPA considers a maximum of 6 CLOS: *CLOS #1* devoted to non-critical and bully applications, *CLOS #2, #3 and #4* for critical applications (CPA supports up to 3 critical applications), and *CLOS #5 and #6:* for squanderer and non-critical applications wasting LLC space, respectively. Experiments were also performed using more CLOS, but results did not improve, thus the presented results only used 6 partitions.

4.5.3. Methodology

To calculate the IPC, MPKI_LLC, HPKI_LLC and LLC occupancy values, we use the following Perf hardware counters: `instructions`, `ref-cycles`, `mem_load_uops_retired.l3_hit`, `mem_load_uops_retired.l3_miss` and `intel_cqm/llc_occupancy/`.

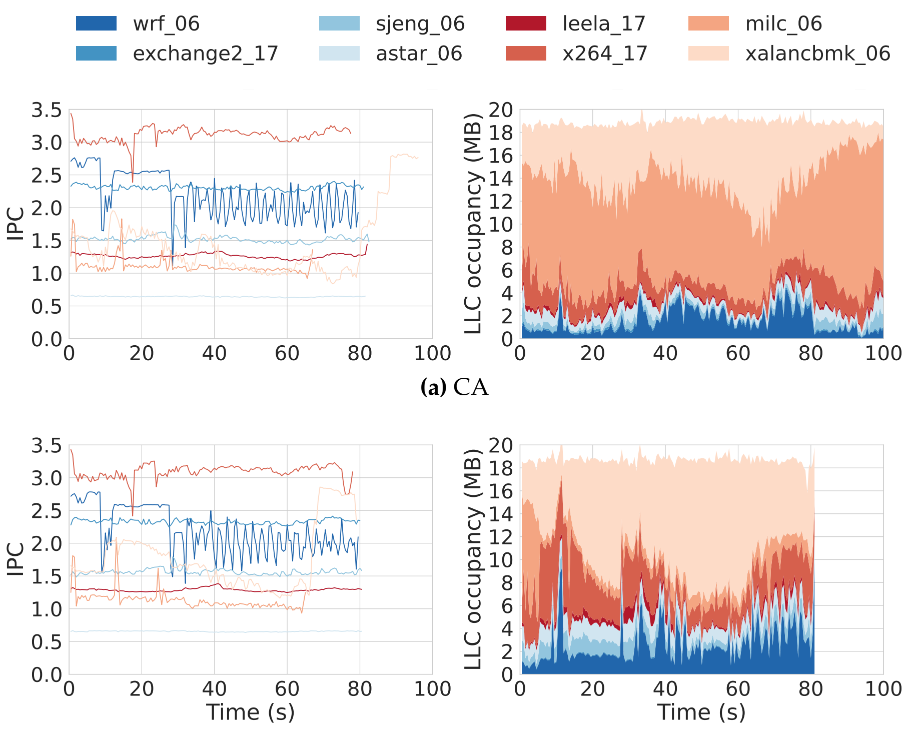
Workload mixes are run until all the applications in the mix have completed a fixed number of instructions. This number corresponds to the number of instructions the application executes when running alone for 60 seconds. When an application reaches this limit, and it is not the last one in the mix to reach such a limit, it is restarted, so the results of the other applications are not skewed by the fact that they have fewer co-runners. Nonetheless, when analyzing the results, only the values of the first execution run of each application are considered. At regular 500ms intervals, the experimental framework reads the performance counters and passes the values to the partitioning algorithm. Each experiment is repeated 3 times, and the average values and the standard deviation for each metric are derived. Results in Section 4.6 are shown within a 95% confidence interval with a margin of error lower than 3%.

4.6 Evaluation

4.6.1. Impact of Newly Identified Behaviors

This section first illustrates the main differences between CPA and CA regarding two key metrics (IPC and LLC occupancy) through the study of an example mix (#20). The mix consists of 1 critical (`xalancbmk_06`), 1 squanderer application (`milc_06`), and 6 non-critical applications. This mix was chosen to help understand why the squanderer behavior is difficult to identify.

Figure 4.8 shows the IPC (before being restarted) and LLC occupancy (for the whole execution) achieved along the execution for each of the 8 applications in the example mix. Figure 4.8a and Figure 4.8b present the results for CA and CPA, respectively. Several observations can be made. Firstly, looking at the X-axis, whose length is bounded by the TT (that is, the execution time of the longest-running application of the mix, `xalancbmk_06`), it can be seen that CPA improves TT significantly, nearly 20%. In this specific mix, CA



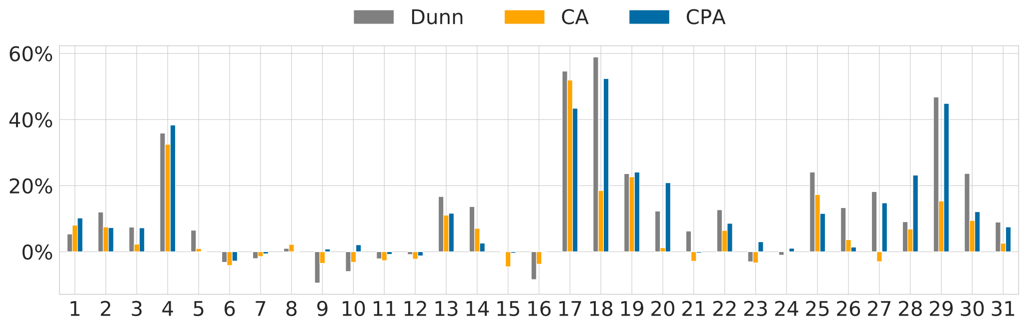
(a) CA

(b) CPA

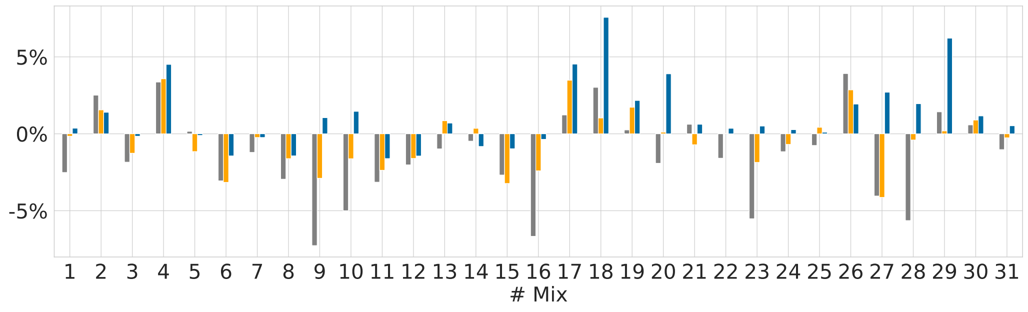
Figure 4.8: Dynamic values of IPC and LLC occupancy of mix #20 under CA and CPA.

achieves similar results to the Linux default NP policy. The main reason is that the behavior of squanderer applications is not correctly detected. Squanderer applications present a high MPKI_LLC. Therefore, CA confuses `milc_06` with a critical application and allocates it into a partition with a greater amount of LLC. However, since squanderer applications present very little reuse, this additional space does not translate into performance gains. This behavior can be observed in the LLC occupancy graph of Figure 4.8a (right side), where `milc_06` occupies a large fraction of the space during nearly the whole execution.

CPA detects and handles this newly identified behavior, so `milc_06` is no longer allowed to occupy such a large LLC space. At the beginning of the execution (by second 5), `milc_06` is identified as a squanderer application because it presents high MPKI_LLC and very low HPKI_LLC. This application shows the same behavior throughout its execution time, so it remains in a separate CLOS with a reduced amount of LLC space. The space released by `milc_06` is taken by `xalancbmk_06`, a critical application that benefits from additional cache space, and the remaining non-critical applications. Notice that gains in TT are due to `xalancbmk_06`, as a non-critical application is barely affected. Looking at the IPC graphs, we can see that even though `milc_06` is assigned much less LLC space by CPA than by CA, its IPC (and execution time) is unaffected. Compared to CA, CPA improves the IPC by nearly 4%.



(a) TT



(b) ANTT

Figure 4.9: TT and ANTT improvement (in %) w.r.t. NP for each workload mix.

4.6.2. Turnaround Time Evaluation

This section compares the turnaround time (TT) and average normalized turnaround time (ANTT) [129] of CPA, CA, and Dunn [1], a state-of-the-art partitioning policy.

Figure 4.9a plots the TT improvement (in percentage) of the studied approaches with respect to NP across 31 8-application workload mixes. CPA improves TT over 40% in three mixes over NP. On average, this improvement is of 11%, which is slightly higher in Dunn (12%) and lower (6%) in CA. An interesting observation is that CPA improves TT considerably with respect to Dunn and CA in those mixes containing a squanderer application, e.g., mix #9 and mix #10. This is because neither Dunn nor CA consider block reuse and LLC occupancy; thus, they are not able to detect behaviors strongly related to these metrics such as those exhibited by squanderer applications; and, as explained above, a wrong classification of squanderer applications leads the system to performance losses. Finally, we would like to remark that CPA manages to reach improvements of over 35% in four mixes.

While TT is primarily a user-oriented performance metric [128], it does not consider the performance losses of an application over isolated execution, which can lead to misleading conclusions. To deal with this fact, we study a complementary metric, ANTT, which should be analyzed alongside TT (see Section 3.5 for further details). Figure 4.9b shows the ANTT improvement (in percentage) achieved by the studied policies over NP for each workload mix executed. As observed, CPA shows the best results, reaching in three

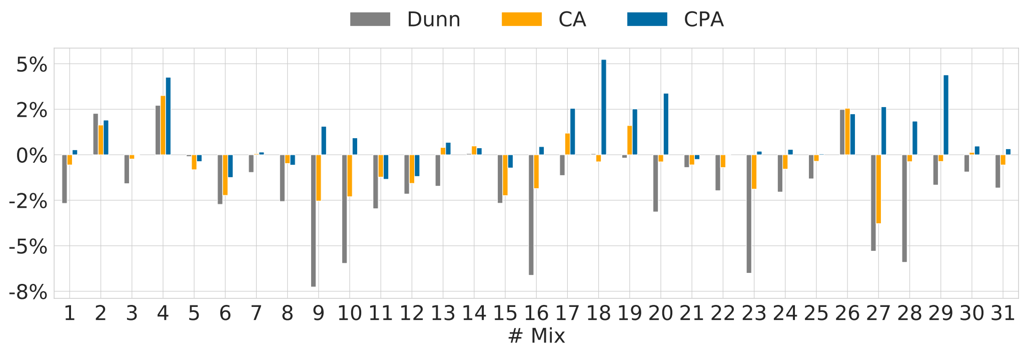


Figure 4.10: IPC (geometric mean) improvement (in %) w.r.t. NP.

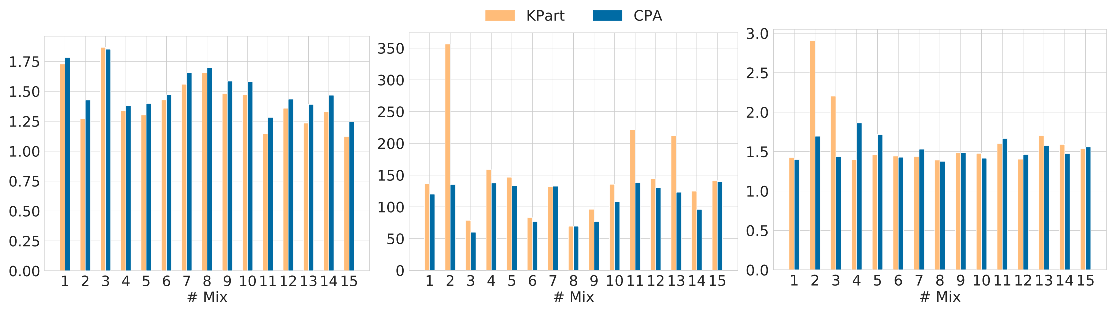
mixes improvements over 4%. In contrast, Dunn degrades this metric by more than 4% in four mixes since it is a more aggressive policy that tries to benefit most those applications showing the highest slowdown (i.e., critical). An important observation is that in those mixes where Dunn outperforms CPA regarding TT, CPA is able to improve ANTT. For instance, CPA manages to reach a TT improvement higher than 50% in mix #18, where Dunn and CA improve by 58% and 20%, respectively. However, CPA outperforms the other two approaches in ANTT by up to 5%. This means that CPA successfully considers the applications' individual performance as well as the overall system performance, and that the gains in the latter are not at the cost of damaging the performance of the co-running applications.

4.6.3. IPC Evaluation

In addition to TT and ANTT, the system throughput is also evaluated in this chapter in terms of IPC (see Section 3.5 for further details).

Figure 4.10 shows the improvement (in percentage) of the IPC geometric mean with respect to NP for each workload mix in the studied policies. Dunn allows the system to achieve good system fairness, however, when problematic behaviors identified in this thesis work are present in the mix, the IPC can drop. The IPC is difficult to sustain in partitioning approaches focused on multi-program workloads mainly because, to deal with system fairness or TT, these partitioning approaches seek to benefit those applications showing an atypical behavior at the expense of damaging the best performing ones. Nevertheless, in spite of this fact, the devised CPA approach properly addresses the newly identified behaviors, improving TT and ANTT while sustaining the IPC or even improving it over 3% in some mixes.

Overall, CPA (and CA but to a lesser extent) manages to maintain and even improve, in some cases, the IPC, whereas Dunn's IPC is worse than NP in most cases. Even though Dunn obtains, on average, a slightly better TT, this improvement should never be at the cost of degrading the system throughput. This principle was first drafted in CA and has been fully tackled in this work.



(a) GeoMean IPC **(b)** TT **(c)** ANTT

Figure 4.11: CPA versus KPart: IPC (geometric mean), TT and ANTT.

Application	KPart		CPA	
	Time (s)	IPC	Time (s)	IPC
sjeng_06	65.5	1.34	38.5	2.28
hmmer_06	60.1	1.45	41.0	2.14
nab_17	71.8	1.22	60.5	1.45
libquantum_06	38.3	2.29	61.5	1.43
tonto_06	41.6	2.10	63.5	1.38
soplex_06	109.3	0.80	65.0	1.35
sphinx3_06	61.1	1.43	85.5	1.03
mcf_06	211.9	0.41	123.0	0.71
TT/IPC(geomean)	**211.9**	**1.23**	**123.0**	**1.39**

Table 4.4: Execution time (s) and IPC the individual applications of mix #13, and TT and IPC (geomean) of the mix.

4.6.4. Comparing CPA with KPart

This section compares our proposal with KPart [46]. This approach groups applications into N clusters, this number ranges from 2 to the number of applications in the mix and each cluster is assigned to a cache partition, which has private ways. In order to run KPart in our experimental platform, we adapted the cache partitioning technique to work with a 20-way cache. We tried the same 31 mixes described in Section 4.5.1, but the KPart framework was only able to run 15 of them, as some SPEC benchmarks are composed of multiple binaries, and supporting that requires important changes in the KPart architecture.

Results for IPC (geometric mean), TT, and ANTT are shown in Figure 4.11. For fair comparison purposes, we also ran the application mixes in CPA until each application of the mix committed 200 billion instructions, like in KPart. As observed, CPA outperforms KPart, on average, by 6% in IPC (geometric mean), 30% in TT, and 6% in ANTT.

To provide further insight into why CPA outperforms KPart, we looked into those workloads where KPart presents a poor performance and found three main reasons: i) KPart allocates multiple applications to 1-way partitions, ii) KPart places problematic and critical applications together, and iii) CPA is triggered when it is really needed (phase changes) and does more precise changes.

For illustrative purposes, we present below a comparative study of mix #13, made up of 5 non-critical applications (`sjeng_06, hmmer_06, nab_17, libquantum_06, tonto_06`), 1 critical sensitive application (`soplex_06`), 1 critical medium application (`sphinx3_06`) and 1 bully application (`mcf_06`). Table 4.4 shows the results of the execution time and IPC obtained for each benchmark of this mix. As it can be observed, the performance (i.e., IPC) of some non-critical applications drops considerably compared to CPA. KPart allocates firstly these applications into the same 1-way CLOS, and then in a CLOS with very little space (1 way/application). Consequently, the partition behaves like a direct-mapped cache, which results in performance degradation. In contrast, CPA does not constrain non-critical applications to such a reduced space, thus it does not present this downside. Another important difference is that KPart is not able to correctly identify bully applications like `mcf_06`. In this example mix, KPart assigns firstly `mcf_06` too little space (1 way), which damages its performance considerably. In the following cache disposal (that is, clustering applications in CLOS and assigning them cache ways), it is placed together with `soplex_06` in a CLOS with a high number of ways (14). However, as shown in the characterization study, problematic applications (`mcf_06` in this example) tend to occupy and waste a high portion of their allocated LLC space. Therefore, placing them in the same cluster as critical applications results in lower performance, as they reduce the space available for the critical ones, without significant performance gains. In the last cache disposal, `mcf_06` is given too much space (9 private ways) that it is not able to use profitably. Notice that CPA mostly uses shared cache ways among CLOS, which, as proved by [1], generally yields a better performance (see Section 4.4.3).

Another difference between KPart and CPA is how often the cache configuration is modified and the number of applications/partitions affected. In this mix, KPart performs a total of 3 cache disposals, compared to CPA, which performs 10 cache configuration updates (following criteria in Table 4.3), and 23 cache adjustments (Step 5 in Algorithm 1). CPA performs more frequent and precise cache configuration updates, which adapts better to behavior changes of the applications.

Consequently, in this example mix CPA improves the IPC (geomean) by 13% since it avoids low IPCs (2 out 8 applications present IPC below 1 in KPart). CPA also outperforms by 61% in TT. We carried out further experiments to estimate the impact of each of the analyzed aspects. We found that TT drops to 53% when phase detection is not applied and to 27% when using 1-way partitions, proving that major performance gains come from the proposed strategy that identifies new specific cache behaviors and performs more precise cache configuration updates.

4.7 Summary

This chapter has presented the work on LLC space management performed in an Intel Broadwell processor with Intel Cache Allocation Technology, one of the first Intel micro-architectures to include this technology.

An exhaustive characterization of the behavior of SPEC CPU 2006 and 2017 applications is performed in terms of the assigned LLC space and interference at its resource. The study has shown that applications behaviors can be classified into five categories: non-critical, sensitive, medium, squanderer, and bully, identified at runtime by monitoring the performance metrics IPC, MPKI_LLC, HPKI_LLC, and LLC occupancy. Applications' LLC behavior is not always the same but may change throughout their execution. The system performance can significantly drop if the LLC space allocated does not correctly adapt to such behavior changes.

In this regard, this chapter has proposed the Critical Aware (CA) and Critical-Phase Aware (CPA) LLC partitioning approaches. CA is a simple approach that only considers the critical and non-critical behaviors, while CPA considers all five behaviors. The major differences between CPA and CA are the following:

- CPA is phase-change driven. This reduces unnecessary checks on the behavior of applications and makes CPA a much more general solution.
- Additional criteria have been introduced to identify the new application behaviors at run-time.
- A higher number of CLOS better fits the higher number of identified behaviors and allows greater control of the LLC space.

Both CA and CPA improve overall system performance by considerably reducing the TT over Linux default behavior. Compared to CA and the state-of-the-art approaches KPart and Dunn, CPA achieves significant performance gains by correctly identifying those applications presenting problematic behaviors and allocating more private space to those presenting sensitive behavior.

The source code of CPA has been made available at `http://hdl.handle.net/10251/143182`.

CHAPTER 5

Non-Inclusive LLC Resource Management

The current trend of server processors is to increase the amount of private cache space (L2 cache) available to the core. As a consequence, the amount of shared L3 cache is reduced considerably. To make better use of the cache space, the design of the cache is no longer inclusive but non-inclusive, and exclusive designs have been adopted in recent processors [138]. However, less space per core to manage and more demanded space (i.e., more traffic since the L3 acts as a victim cache) by applications means that fierce competition for cache space will take place.

This chapter analyzes the cache requirements of SPEC CPU2006 and SPEC CPU2017 applications[1] in non-inclusive caches compared to inclusive caches, as well as the inter-application interference that rises due to the reduced L3 cache space. The results of this study are used to design Cache-Poll, a cache partitioning strategy aimed at improving system throughput in non-inclusive caches by minimizing cache pollution.

5.1 Motivation

Cache management is more critical in processors with non-inclusive L3 caches due to two main reasons that arise by design. On the one hand, they present smaller capacity per core both in the L3 and in the entire cache hierarchy than with inclusive L3 caches. This means that there is less flexibility in assigning L3 cache space to applications. On the other hand, the fact that the L3 cache is non-inclusive implies that a significant number of the evicted L2 cache blocks (regardless if they are dirty or clean blocks) are written back to the L3 cache.

To illustrate this claim, we measured the L2 writebacks in both Intel Broadwell and Skylake-X processors, implementing an inclusive and non-inclusive L3 cache, respec-

[1]Since there are applications whose name appears in both suites, from now on, the suffix `_06` and `_17` will be added to specify the corresponding suite.

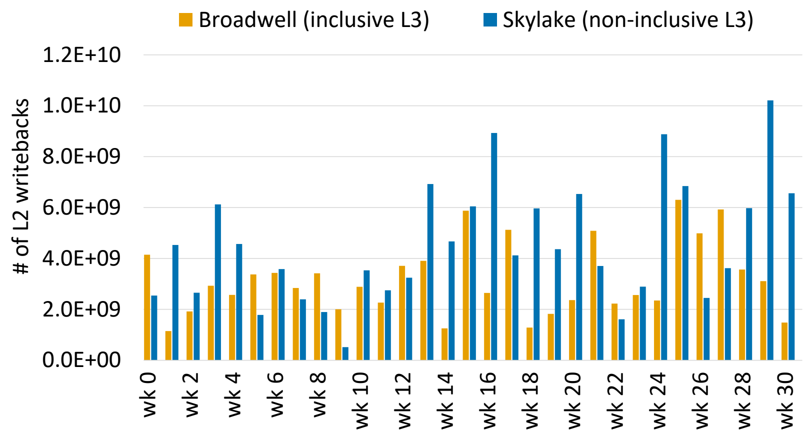


Figure 5.1: L2 cache writebacks in 31 8-application workloads executed in an Intel Broadwell and Skylake-X processors.

tively. Figure 5.1 shows the results for 31 randomly generated 8-application workloads. Despite that the L2 cache is four times larger, the total number of L2 writebacks to the non-inclusive L3 cache of the Skylake-X processor is much higher ($3\times$ to $4\times$) than that of the Broadwell.

This increased traffic implies that it is even more critical than in inclusive caches to make efficient use of the smaller L3 cache space. For this purpose, we focus on minimizing the effect known as *cache pollution*. The term *pollution* has been used in the past to refer to inaccurate prefetch requests (i.e., useless prefetches) and addressed in different ways [2, 3]. In this work, we extend its meaning to refer to any action that allocates new blocks into the cache that are never (or scarcely) referenced. Under this umbrella, pollution can arise from different sources. In addition to useless prefetches, we identify other causes of cache pollution; misspeculated loads, that is, load instructions that are executed at the shadow of a misspeculated branch; and poor locality, which refers to blocks that are brought by demand but not (or scarcely) referenced again. Notice that although the latter blocks are not, strictly speaking, polluting the cache, we consider these blocks under the umbrella since they introduce a certain type of pollution by replacing other useful blocks.

5.2 Background

The on-chip cache hierarchy has evolved in the last processor generations, like Intel Xeon Scalable [101] starting with the Skylake-X architecture, from inclusive to non-inclusive L3 caches.

Inclusive L3 caches keep copies of all the blocks stored in the private L2 caches. To be effective, its size needs to be much larger than the storage capacity of all the L2 caches. When a block is fetched from main memory, it is filled up both in the L2 and L3 caches (see Figure 5.2). Then, when the block is evicted from the L2 cache, it is written back to the L3 only in case it has been modified.

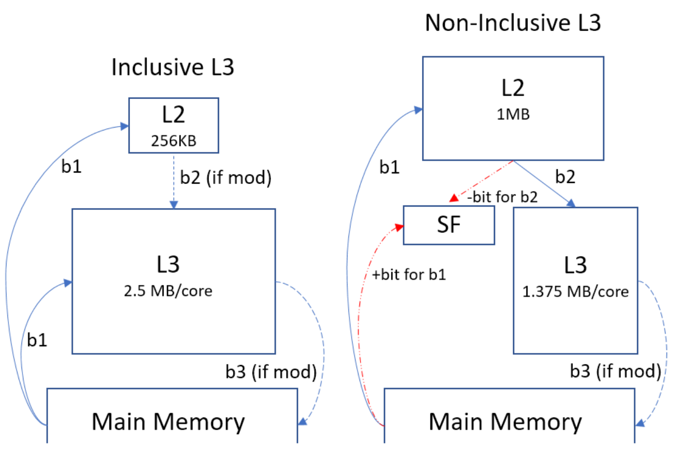


Figure 5.2: Difference between inclusive and non-inclusive L3 caches when a miss occurs both at the L2 and L3 caches.

In non-inclusive L3 caches, a block in the L2 cache may or may not be present in the L3 cache. Different implementations are possible. Figure 5.2 (right side) shows the scheme deployed in the Intel Skylake-X architecture [101]. Upon an L3 cache miss, the requested block is fetched from memory and filled into the L2 cache. The L3 cache serves as a victim cache of the L2 caches (i.e., upon eviction from the L2, most blocks are written back to the L3 regardless of whether they have been modified or not) and for prefetching purposes. However, it does not act as a pure victim cache, but the design allows to drop and not write back to the L3 cache those evicted L2 blocks that are deemed to be less likely to be reused shortly [139]. In this processor, to help guarantee cache coherence, the Snoop Filters extend the L3 cache directory to keep track of the blocks stored in the L2 caches.

5.3 Characterizing L3 Cache Behavior

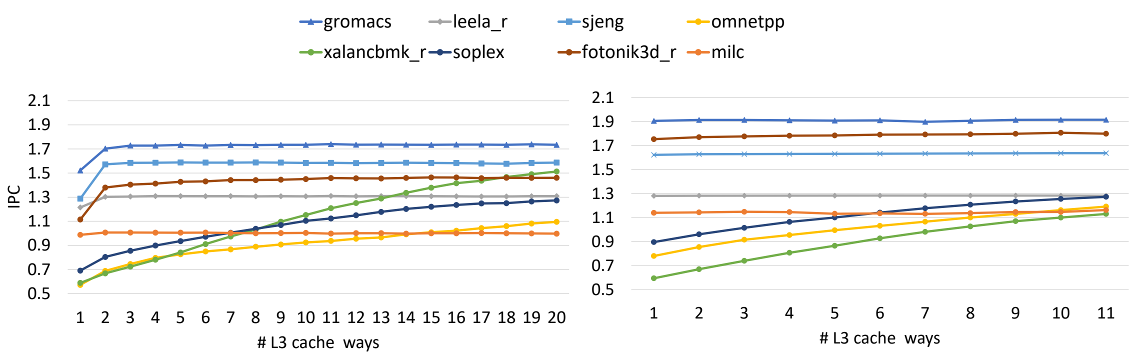
Ideal cache sharing would assign additional cache space to an application only if it translates into performance improvements without degrading the performance of the co-runners.

This section first characterizes how the available L3 cache space impacts the performance of the studied applications. Then, we study the types of cache pollution introduced by the applications and how it slows down the execution time.

5.3.1. Space Requirements in Non-Inclusive vs. Inclusive Caches

The new cache hierarchy organization (larger L2 cache and smaller non-inclusive L3 cache) deployed in the Intel Scalable family has changed the L3 cache needs of applications to achieve maximum performance compared to previous processor generations.

Figure 5.3 shows how the achieved performance (i.e., IPC) evolves, both in an Intel Broadwell processor (Figure 5.3a) and in an Intel Skylake-X processor (Figure 5.3b), as the num-



(a) Intel Broadwell **(b)** Intel Skylake-X

Figure 5.3: Performance of individual applications with an increasing number of ways in Intel Broadwell and Skylake-X processor.

ber of cache ways increases from 1 to the entire cache (20 and 11 ways, respectively) for eight representative applications when running in isolation. Three main observations can be made.

First, we distinguish two main application behaviors, depending on whether the applications' performance improves as the number of cache ways increases. The latter applications will be referred to as **L3 insensitive**. Applications showing this behavior are `gromacs_06`, `leela_17`, `sjeng_06`, `fotonik3d_17` and `milc_06`.

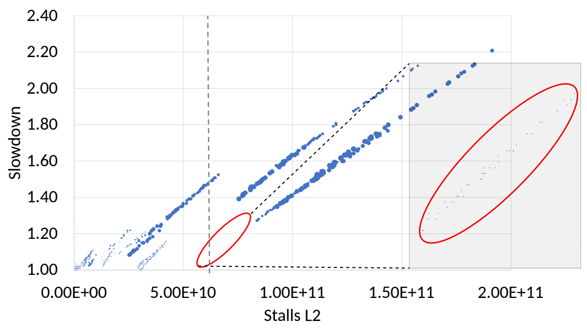
Second, notice that only one cache way is enough for L3 insensitive applications to achieve maximum performance in the Skylake-X processor when running alone. The reason why they have such little space is twofold. On the one hand, the working set fits better in the larger L2 cache. On the other hand, by design, since the L3 is non-inclusive, it keeps a few replicas of L2 cache blocks. On the contrary, in the inclusive L3 cache of the Intel Broadwell, applications need to replicate their working set both in the L2 and L3 caches; therefore, the L3 space cannot be limited as much.

Third, the previous rationale means that there will be fewer accesses to the L3 cache when running alone, especially in L3 insensitive applications, and therefore, they will achieve higher IPC in the Intel Skylake-X. However, high cache-demanding applications (from now on referred to as **L3 sensitive**), like `xalancbmk_17`, achieve higher performance in the Broadwell processor when occupying the whole (20 ways) larger 20MB L3 cache.

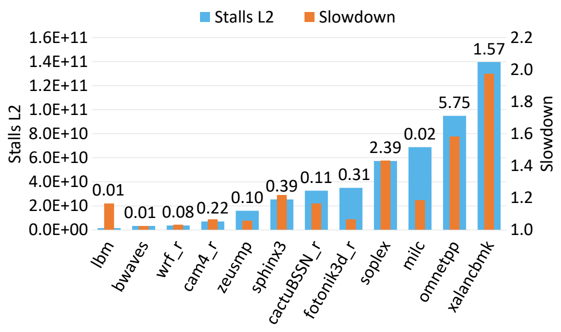
After analyzing the impact of the non-inclusive L3 cache, from now on, the remainder of the chapter will focus on the Intel Skylake-X processor.

5.3.2. Estimating Slowdown in Non-Inclusive Caches

A key step in any sharing policy is to identify the applications whose overall performance benefits most from additional cache space since the L3 cache space is shared among multiple co-runners, and the final goal is to maximize overall performance. That is, identify those (L3 sensitive) applications that experience higher slowdown (over individual exe-



(a) $Stalls_{L2}$ v.s slowdown (size = $HPKI_{L3}$). Zoomed area shows tiny points introducing pollution



(b) Example workload (number of top of bars refers to $HPKI_{L3}$)

Figure 5.4: Study of the correlation of $stalls_{L2}$ and $HPKI_{L3}$ with slowdown of individual applications executed in multi-program execution.

cution) when the available cache space is reduced, which is challenging. The challenge lies in that this estimate must be done at run-time in multi-program execution. This section deals with this challenge in non-inclusive caches.

Previous works [1, 46, 55] and the approach presented in Chapter 4, have proved that L3 sensitive applications can be detected with metrics such as L3 cache misses and processor stalls due to these misses in inclusive caches. We analyzed a wide set of metrics reported in the literature (e.g., number of L3 misses per kilo instruction committed) and found that the number of processor stalls due to L2 misses ($stalls_{L2}$)[2] is the metric that best correlates in non-inclusive L3 caches.

However, as pointed out by [55], some *aggressor* L3 insensitive applications experience a high number of stalls and could be mistaken as L3 sensitive applications if only relied on this metric. Therefore, additional metrics are required to allow distinguish this situation. In this regard, CPA, presented in Chapter 4, also considered the hits in the L3 cache as a good proxy to distinguish *aggressor* L3 insensitive applications from sensitive applications.

Figure 5.4a shows the slowdown and the $stalls_{L2}$ of individual applications[3] when executed with 11 co-runners in 12-application workloads (i.e., 12 points are presented for each workload). A total of 1128 points are plotted. The point size in the graph grows linearly according to the hits per kilo instructions performed at the L3 cache ($HPKI_{L3}$). Notice that a good correlation can be observed (0.8704) between $stalls_{L2}$ and slowdown in the Skylake server processor. However, it is 17% lower than that found by Selfa *et al.* in the Haswell processor [1]. This can be observed in the plot where points do not form a unique tendency line but follow several parallel trends. However, note that the point size varies among the different tendency lines for a given value of $stalls_{L2}$. That is the value of the $HPKI_{L3}$ differs. This can be seen in the area between 5.0×10^{10} and 9.0×10^{10} (marked with a red oval and zoomed in grey color) with minuscule points (i.e.,

[2]Performance counter cycle_activity.stalls_l2_miss
[3]SPEC CPU2006 and SPEC CPU2017

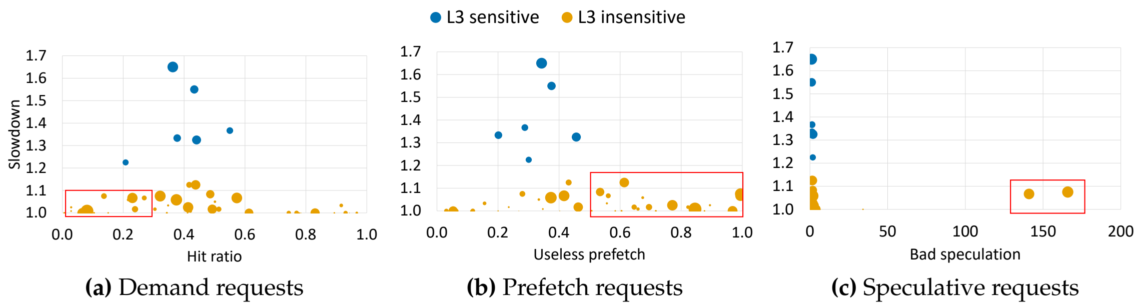


Figure 5.5: Individual characterization of SPEC CPU2006 and SPEC CPU2017 applications under limited cache space (2 ways or 3MB) considering demand, prefetch, and speculative requests. Each point is sized according to its $MPKC_{L2}$.

almost zero $HPKI_{L3}$). It can also be appreciated that there are at least two applications with the same amount of stalls but with opposite cache sensitivity behavior. This can be observed looking at the vertical line crossing at 6.0×10^{10} $stalls_{L2}$, which cuts across applications suffering around a slowdown of 1.1 (minuscule points located in the red oval with $HPKI_{L3}$ close to zero) and 1.5 (bigger points).

Figure 5.4b illustrates this claim through an example. This plot depicts the $stalls_{L2}$ (blue bar, left y-axis), the slowdown (orange bar, right y-axis), and the $HPKI_{L3}$ (number on top of the bar) obtained by each application that makes up the workload. As it can be observed, `milc_06` has similar stalls to `soplex_06` (and even higher), but `milc_06` experiences less slowdown than `soplex_06` (1.19 versus 1.43). However, `soplex_06`'s $HPKI_{L3}$ is much higher than `milc_06`'s, which is negligible. Therefore, we can conclude that $stalls_{L2}$ alone is not a good proxy for performance to detect L3 sensitive applications, but additional metrics like the $HPKI_{L3}$ are needed.

5.3.3. Identifying Cache Pollution

Applications polluting the cache introduce inter-application interference on performance since cache lines of other applications are forced to evict, which makes them slow down their execution. Therefore, the pollution effect caused by L3 insensitive applications needs to be minimized to achieve the best performance. This section aims to identify applications introducing pollution from three main sources: useless prefetches, loads executed in the shadow of a misspeculated branch, and poor locality.

Requests from all the applications compete for L3 cache space. Many works in the past used the misses per kilo instructions (MPKI) as a metric to evaluate both the cache performance and identify memory-intensive applications. Since applications execute instructions at very different rates (e.g., IPC varying from 0.4 to 3.4), then it makes more sense to use the misses per kilo processor cycle (MPKC) at the L2 ($MPKC_{L2}$) instead of the MPKI as the metric to compare the number of L3 cache accesses performed by the applications.

The key goal of the experiments carried out in this section is to identify cache-polluting applications that do not (or slightly) suffer from a reduced cache space. To this end, experiments were conducted by limiting the cache space to only two cache ways (i.e., 3MB). Figure 5.5 plots the relationship between the slowdown suffered by 50 applications over full cache ways (i.e., 16.5MB) and three main pollution-related metrics: useless prefetches, bad speculated loads and hit ratio. Applications with a slowdown lower than 1.2 (L3 insensitive) are colored in yellow. Points colored in blue correspond to L3-sensitive applications that require a high amount of space for performance; thus, they cannot be constrained to a few cache ways. In the three graphs, the point size is proportional to the $MPKC_{L2}$.

Figure 5.5a shows how the L3 hit ratio relates to the slowdown. The analysis focuses on big points (i.e., high access rate to the L3 cache) colored in yellow, that is, exhibiting a low hit ratio (e.g., below 0.3 in the red box). These applications are `bwaves_17`, `fotonik3d_17` and `lbm_06`. Even though these applications experience a small slowdown, they introduce pollution in the cache since most of them are not reused again.

Regarding useless prefetches, a similar reasoning can be done. Figure 5.5b shows the results. Useless prefetches have been quantified as the ratio of prefetch accesses that miss in the L3 cache over the total cache accesses made to the L3 cache using the performance events indicated in Table 5.1. Applications introducing more pollution are those represented as "big points", as they bring much useless data into the L3 cache but experience a low slowdown (see red box). These applications are `bwaves_17`, `fotonik3d_17`, `lbm_06`, `lbm_17`, `mcf_17`, `nab_17` and `namd_17`.

Finally, we quantify the pollution introduced by loads executed at the shadow of a mispredicted branch labeled as bad speculation [140]. Figure 5.5c shows the results. Bad speculation has been quantified as the total number of demand requests that miss the L2 cache over the number of retired loads that miss the L2 cache (1 means there is no bad speculation). Notice that most applications present a similar value, except for `lbm_06` and `lbm_17`, which show an extremely high value. Thus, we can conclude that bad speculation is the least critical of the three studied cache-polluting sources across all the studied applications.

5.4 Cache-Poll Approach

This section presents Cache-Poll, an L3 non-inclusive cache management approach that efficiently detects applications causing pollution an reduces its space to a minimum, leaving room for L3 sensitive applications. Algorithm 2 shows the pseudo-code of Cache-Poll. The algorithm consists of four main steps, which are explained next.

5.4.1. Classifying Applications

In the first step (lines 1 to 4 of Algorithm 2), performance counters (events) are read with Perf [115] each execution quanta (e.g., 500ms), which are used in subsequent steps to compute the required performance metrics. Table 5.1 shows the relationship between performance metrics and events.

Algorithm 2 Cache-Poll pseudo-code

```
———— 1. Read and compute metrics ————
for all applications do
    Read performance events and compute metrics (see Table 1)
end for
———— 2. Classify applications ————-
for all applications do
    —- A) DETECT DEGREE OF SENSITIVITY —-
    if high $stalls_{L2}$ & high $HPKI_{L3}$ then
        L3 sensitive behavior
    else if low $stalls_{L2}$ & low $HPKI_{L3}$ then
        L3 insensitive behavior
    else
        Mild L3 sensitive behavior
    end if
    —– B) DETECT CACHE POLLUTERS: L3 insensitive —–
    if high $MPKC_{L2}$ then
        if (low $HPKI_{L3}$ & low $hitRatio$) | (high $fractionPF$ & high $uselessPF$) | (high $badSpeculation$) then
            L3 insensitive behavior
        end if
    end if
end for
———— 3. Assign applications to CLOS ————
for all applications do
    if L3 insensitive then
        Assign to CLOS #1 with 2 ways
    else if L3 sensitive then
        Assign to CLOS #2 with full ways
    else
        Assign to CLOS #3 with 3/4/full ways*
    end if
end for
——- 4. Refine L3 sensitive CLOS space distribution ———
for all L3 sensitive applications do
    if low $stalls_{L2}$ w.r.t maximum $stalls_{L2}$ then
        Assign to CLOS #4 with 7 ways
    end if
end for
```

In the next step, applications are classified according to the sensitivity their performance shows to the assigned cache space in order to allocate them the correct cache share. This is done at the start of the execution (after some warm-up intervals) for all applications to identify their initial behavior. After this point, applications' behavior will only be checked when they experience a *phase-change* which minimizes overhead. That is when their performance (i.e., IPC) trend changes. To identify execution phase changes, we leverage ICOV's method [136]. Classification is performed in two major steps:

1. **Detect degree of sensitivity** (lines 7 to 14). In this first step, the level of sensitivity is explored using the $stalls_{L2}$ metric together with the $HPKI_{L3}$.

2. **Detect cache pollution** (lines 15 to 22). In this second step, different checks are performed concerning the three main sources of cache pollution discussed in previous sections.

Metric	Performance event(s)
$stalls_{L2}$	cycle_activity.stalls_l2_miss
$HPKI_{L3}$	inst_retired.any mem_load_retired.l3_hit
$hitRatio$	mem_load_retired.l3_hit mem_load_retired.l3_miss
$fractionPF$ $uselessPF$	offcore_response.all_pf_data_rd.l3_miss.any_snoop offcore_response.all_pf_data_rd.l3_hit.any_snoop offcore_response.demand_data_rd.l3_miss.any_snoop offcore_response.demand_data_rd.l3_hit.any_snoop
$badSpeculation$	l2_rqsts.demand_data_rd_miss mem_load_retired.l2_miss

Table 5.1: Metric-performance events relationship, for each of the metrics used in Cache-Poll. Note $fractionPF$ and $uselessPF$ are computed using the same performance events.

Metric	Level	Threshold
$stalls_{L2}$	high	>Q3(rolling_window, N=10)
	low	<Q1(rolling_window, N=10)
$HPKI_{L3}$	high	>AVG(rolling_window, N=10)
	low	<1
$MPKC_{L2}$	high	>AVG(rolling_window, N=10)
$hitRatio$	low	<0.2
$fractionPF$	high	>80
$uselessPF$	high	>50
$badSpeculation$	high	>2

Table 5.2: Thresholds empirically determined for our experimental platform for the metrics and levels used in Cache-Poll.

Performing the classification in two rounds allows to detect those applications that may present a considerable degree of sensitivity but introduce more pollution than performance gains are obtained when allocating additional space (see Section 5.3).

Algorithm 2 uses two thresholds (low and high) to estimate the sensitivity levels of the different metrics. Their value has been empirically established after performing a high number of experiments. Table 5.2 shows the threshold values used in the experimental evaluation.

5.4.2. Distributing the L3 Cache Space

Once the applications have been classified, the L3 cache space is distributed accordingly. Since Intel CAT partitions the cache at a per-CLOS basis, applications must be first grouped into CLOS before dividing the L3 cache space. L3 cache partitioning in Intel Skylake-X represents a significant challenge compared to previous architectures since it has lower cache capacity (16.5 MB compared to 20 MB) and less but larger cache ways (11 1.5 MB ways compared to 20 1 MB ways). This lower granularity provides less flexibility to partition the L3 cache space. Additionally, the reduced number of ways (11) limits the scalability of the number of co-runners if private ways are devoted to individual applications. Therefore, having small cache partitions to isolate applications like in state-of-the-art approaches [53, 46, 55] is no longer suitable. However, as shown in Section 5.3.1, L3 insensitive applications have fewer cache space requirements, and thus, they can be placed together in a small cache partition.

At the start of the execution, Cache-Poll places all the applications in the same CLOS (#1) with full cache ways. From then on, when an application is identified as L3 sensitive, CLOS #1 space is reduced to 2 ways, and the identified application(s) are placed in a separate CLOS (#2) with all the available cache space (11 ways). That is, two ways are being used by both CLOS. We tested different cache sizes for CLOS #1 and found that 1-way partition incurred many space conflicts and evictions in the L3 cache directory [141], harming performance severely. Instead, 2-way partitions (adding more cache ways to CLOS #1 resulted in no performance gains) solved this issue and allowed L3 sensitive applications to have the maximum possible private space. If a mild L3 sensitive application is detected, Cache-Poll places it in CLOS #3 with some extra space over the 2-way CLOS #1. CLOS #3 is sized depending on the number of co-running L3 sensitive applications: i) all the cache ways if there are no sensitive applications, ii) 4 ways if there are less than three sensitive applications, and iii) 3 ways otherwise. In this way, more or less space is given depending on the sensitivity of the co-runners since L3 sensitive applications benefit most from additional L3 space.

5.4.3. Refining L3 Cache Space Distribution

Finally, in the last step of Cache-Poll (lines 32 to 37 of Algorithm 2), further refinements are performed to tune the cache space distribution to the cache demands of the applications. More specifically, a more refined space distribution is performed among L3 sensitive applications since we observed that not always the most sensitive application occupied the largest L3 cache share. That is, some less sensitive applications presented a higher L3 access rate (i.e., high $MPKC_{L2}$) and, thus, occupied more cache space. To deal with this fact, L3 sensitive applications presenting less than 60% of the $stalls_{L2}$ of the most L3 sensitive application are placed in a separate CLOS (#4) with a reduced amount of space (7 ways).

5.5 Experimental Setup

Both the methodology followed to carry out the experiments and run the workload mixes, and the software manager to conduct them were the same as those used in Chapter 4 focusing on inclusive caches. The experiments have been conducted in the Intel Skylake processor described in Section 3.1.

L3 sensitive applications
mcf_06, omnetpp_06, omnetpp_17, soplex_06, xalancbmk_06, xalancbmk_17

L3 insensitive applications
astar_06, bwaves_06, bwaves_17, bzip2_06, cactusADM_06, calculix_06, cam4_17, dealII_06, deepsjeng_17, exchange2_17, gamess_06, gobmk_06, gromacs_06, h264ref_06, hmmer_06, libquantum_06, perlbench_06, parest_17, mcf_17, imagick_17, leslie3d_06, lbm_06, lbm_17, leela_17, nab_17, namd_06, namd_17, povray_06, povray_17, perlbench_17, sjeng_06, tonto_06, wrf_06, wrf_17, blender_17, cactuBSSN_17, fotonik3d_17, GemsFDTD_06, roms_17, sphinx3_06, zeusmp_06, milc_06, perlbench_06

Table 5.3: Classification of SPEC CPU2006 (_06) and SPEC CPU2017 (_17) applications.

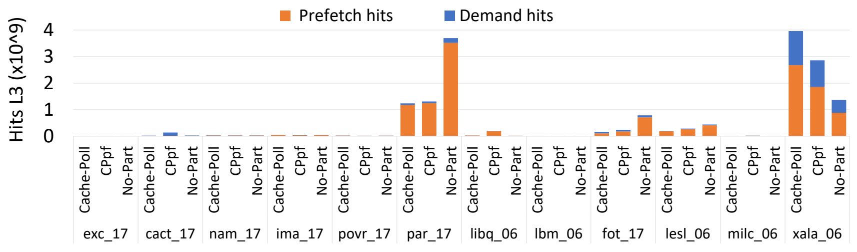
Mix #	$\sum$**L3 insensitive** $MPKC_{L2}$	$\sum$**L3 sensitive** $MPKC_{L2}$
1 - 17	$\approx$150	$\approx$ from 20 to 60
18 - 35	$\approx$200	$\approx$ from 20 to 60
36 - 54	$\approx$250	$\approx$ from 20 to 60

Table 5.4: Computation of the L3 cache access rate (in misses per kilo cycle) of the 54 12-application workloads.

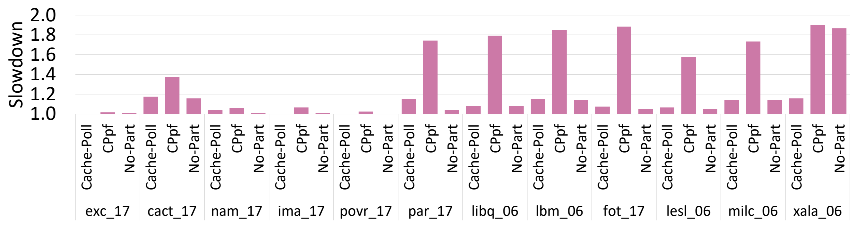
5.5.1. Workload Mixes

To evaluate Cache-Poll, we have randomly generated 54 12-application (as many applications as cores in the processor) workloads with the SPEC CPU2006 (reference input data) and SPEC (rate) CPU2017 benchmark suites. We have used other workload mixes than those used to evaluate the work carried out in the inclusive LLC as applications' level of cache sensitivity differs in the processor with the non-inclusive caches (mainly due to the hierarchy including caches with different sizes and policies). On top of this, the Skylake processor has more cores (12 compared to 8), therefore, mixes with a higher number of applications needed to be generated.

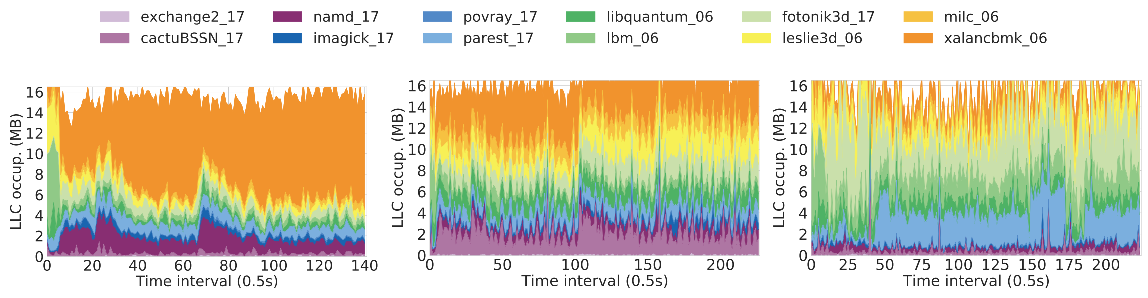
Table 5.3 classifies the studied applications as L3 sensitive and insensitive depending on whether their performance improves or not with additional cache space, respectively. The designed workload mixes include both sensitive and insensitive applications and have been generated considering different levels of L3 cache access rates ($MPKC_{L2}$), shown in Table 5.4. Mixes are grouped in three main groups (#1-#17, #18-#35, and #36-#54) depending on the L3 insensitive $MPKC_{L2}$, and within each group, there are sorted in ascending order of L3 sensitive $MPKC_{L2}$. For a given mix, the $MPKC_{L2}$ level is computed as the sum of the (sensitive or insensitive) applications' $MPKC_{L2}$, obtained in individual execution. To create cache pollution, the $MPKC_{L2}$ for the L3 insensitive applications ranges from 150 to 250, and for sensitive applications from 20 to 60. Notice that this design forces the algorithm to focus on dealing with the huge interference introduced by insensitive applications.



(a) L3 cache hits



(b) Slowdown



(c) Cache-Poll occupancy **(d)** CP$_{pf}$ occupancy **(e)** No-Part occupancy

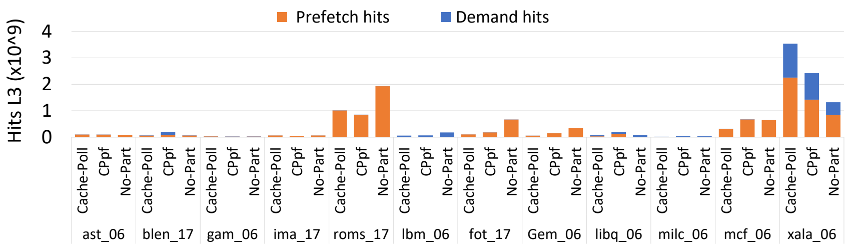
Figure 5.6: Comparison of the L3 cache hits, slowdown and occupancy experienced by the 12 applications of mix 36.

5.6 Evaluation

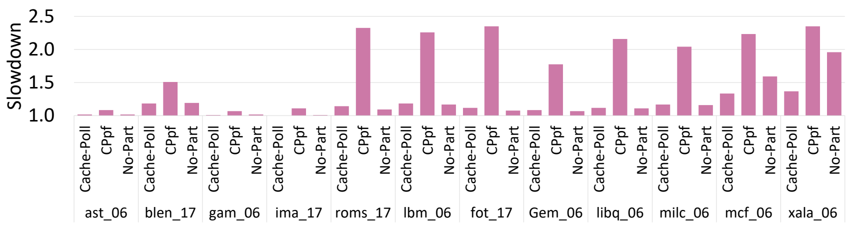
5.6.1. Enhancing Performance by Containing Pollution

Cache-Poll is aimed at containing pollution in a few L3 cache ways so that L3 sensitive applications benefit from this freed cache space, improving the overall performance. This section evaluates how Cache-Poll addresses cache pollution by performing an efficient cache space distribution.

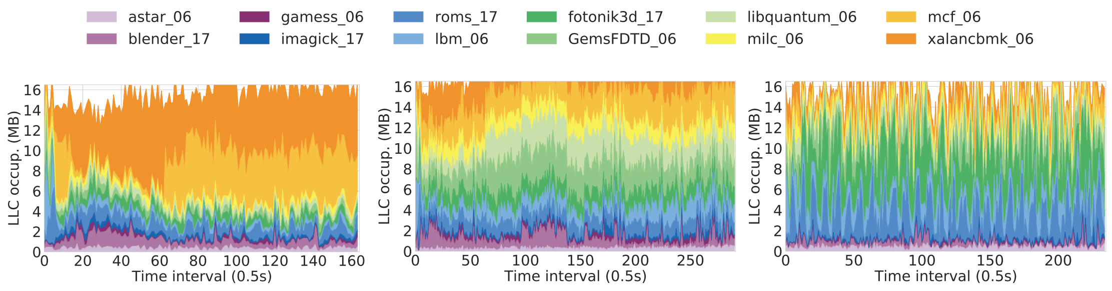
To illustrate how Cache-Poll contains pollution, we compare its performance against a state-of-the-art approach CP$_{pf}$ [53] and No-Part (i.e., default Linux OS) using two 12-application workloads (mixes 36 and 44). These mixes showcase how Cache-Poll efficiently contains pollution and adapts its partitioning scheme to the different execution phases. Performance results are detailed for each application of the workload.



(a) L3 cache hits



(b) Slowdown

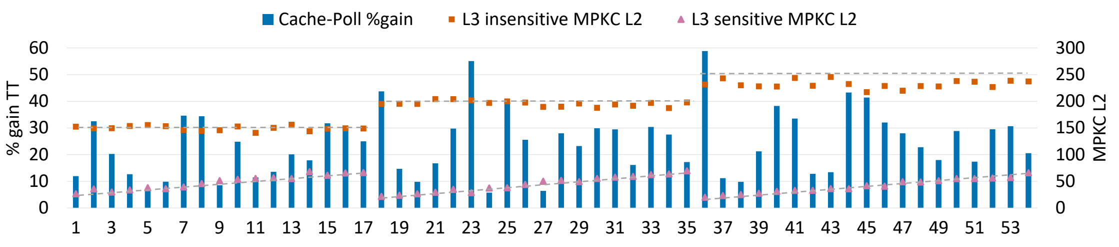


(c) Cache-Poll occupancy **(d)** CP_{pf} occupancy **(e)** No-Part occupancy

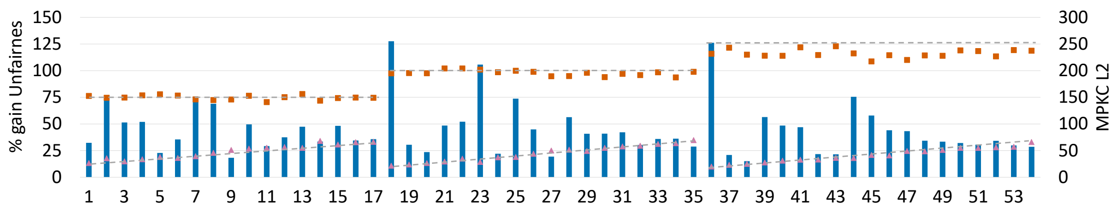
Figure 5.7: Comparison of the L3 cache hits, slowdown, and occupancy experienced by the applications of mix 44.

Figure 5.6 shows the results for mix 36. In the top row of graphs, three bars are plotted for each application, corresponding to results obtained for the three evaluated approaches. Figure 5.6a shows the L3 demand and prefetch hits, and Figure 5.6b shows the slowdown. As it can be observed, Cache-Poll is the approach showing the highest number both of L3 demand and prefetch hits for the L3 sensitive application `xalancbmk_06`, that is, the one experiencing the highest slowdown. This increase in L3 hits allows `xalancbmk_06` to reduce its slowdown from 87% to 18% compared to No-Part. The reason for such slowdown reduction is that Cache-Poll controls and isolates pollution in a few cache ways. Due to this fact, more room is made available for `xalancbmk_06` (see Figure 5.6c) that is used both for demand and prefetch requests. In contrast, `parest_17`'s available space is reduced considerably in Cache-Poll compared to No-Part, reducing the L3 prefetch hits. Even so, this does not translate into important performance losses, unlike with L3 sensitive applications.

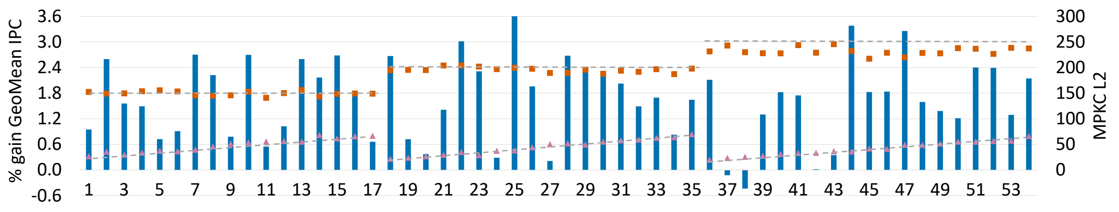
Notice that, even though `xalancbmk_06` has more space in CP_{pf} than in No-Part (see Figures 5.6d and 5.6e), its slowdown does not improve. The reason is that CP_{pf} places



(a) Turnaround Time



(b) Unfairness



(c) GeoMean IPC

Figure 5.8: Performance and fairness results for Cache-Poll compared against No-Part.

`xalancbmk_06`, classified as non-prefetching sensitive (NPS), together with other NPS applications in the same partition (with `exchange2_r`, `cactuBSSN_17`, `namd_17`, `imagick_17`, `povray_17`). Additionally, due to the high amount of prefetching sensitive (PS) applications (the remaining six applications of the mix) and the partitioning scheme of CP_{pf}, only six ways are available for NPS applications. On the contrary, Cache-Poll assigns more private space to `xalancbmk_06` (9 private ways) since the L3 insensitive applications are placed together with the cache-polluting applications, avoiding interfering with sensitive applications. This translates into important turnaround time benefits: Cache-Poll achieves a speedup by 58.8%, while CP_{pf} does not manage to improve the mix's turnaround time.

Similar reasoning can be applied to mix 44 presented in Figure 5.7. In this mix, Cache-Poll identifies two L3 sensitive applications, `xalancbmk_06` and `mcf_06`. However, in this case, `mcf_06` does not exhibit the same behavior across the entire execution, but it shows L3 sensitive behavior between intervals 5 and 10 and from interval 60 until it finishes its execution. Since Cache-Poll detects execution phase-changes, it can adapt dynamically its cache partitioning scheme to suit the cache requirements in each phase. In contrast, CP_{pf} performs a more general classification based on the benefit applications experience from hardware prefetchers, and thus, it does not consider run-time cache requirements.

Compared to No-Part, the achieved performance gains by Cache-Poll are by 43% while CP$_{pf}$ results in around 15% performance loss.

In summary, Cache-Poll has proved effective in containing cache pollution and providing the largest amount of space to L3 sensitive applications. In contrast, mainly due to design reasons, CP$_{pf}$ has not proved to be effective when executing a high number of applications and to contain cache pollution effectively, thus harming the performance of L3 sensitive applications.

5.6.2. Performance and Fairness

This section evaluates the performance of Cache-Poll in terms of turnaround time (TT) and geomean IPC, and unfairness. To compute unfairness, first, the inverse of the slowdown that each application experiences in multi-program execution is computed. Then, the standard deviation is obtained and divided by the average [142].

As mentioned in Section 5.5.1 workload are classified in three main groups (#1-#17, #18-#35, and #36-#54) according to the sum of $MPKC_{L2}$ of the cache insensitive applications. To help the analysis, the sum of the $MPKC_{L2}$ of both cache-sensitive (purple points) and insensitive applications (orange points) have been plotted. Each of the studied levels is marked with a grey dashed line.

Figure 5.8a presents the TT gains over No-Part for the 54 studied mixes. Cache-Poll achieves the best performance, in general, when there is a high difference between the two plotted grey dashed lines ($\sum MPKC_{L2}$ of insensitive and sensitive applications) like in mixes 18, 23, 26, and 44. This is mainly due to a high number of cache-polluting applications making competition for space harder for demand requests of sensitive applications. The studied workloads include up to 3 L3 sensitive applications, which is reasonable considering the low frequency (6 out of 50) of this type of application across all the SPEC CPU. Nonetheless, we checked Cache-Poll's behavior with a high number (up to 7) of L3-sensitive applications. In this case, performance gains over No-Part decrease smoothly with the number of sensitive applications despite performance gains still remaining significant (4% with seven sensitive applications). This is reasonable since the more L3-sensitive applications, the less room for (insensitive) polluting applications; thus, the magnitude of the problem decreases. Regardless of the considered level of L3 sensitive and/or insensitive applications, Cache-Poll effectively detects the identified cache polluting behaviors, partitioning the cache so that L3 sensitive applications are provided with most of the space, and insensitive and polluting applications are contained in a few cache ways, obtaining significant TT reductions.

Reducing TT translates, in general, into significant unfairness improvements. Figure 5.8b shows the results. In some mixes like 18 and 36, unfairness is reduced more than double over No-Part. Finally, Figure 5.8c shows that TT and unfairness are not improved at the cost of IPC. In contrast, the geomean of IPC is improved up to 3.6% and on average by 1.45%.

5.7 Summary

L3 cache pollution rises due to the smaller L3 cache space available in modern processors and the higher L2-L3 traffic induced by the non-inclusive L3 cache. This effect is especially significant when the processor runs a high number of applications stressing the L3 cache. Consequently, pollution needs to be addressed to improve processor performance. This research has identified performance metrics to detect sensitive applications to the cache space and measured three main types of L3 cache pollution at run-time.

This chapter has presented Cache-Poll that, based on the *polluting* behavior and the cache requirements of the co-running applications, distributes the L3 cache space at run-time among applications. Cache-Poll takes benefit of the non-inclusive L3 design by leaving little room for cache-insensitive and polluting applications. This implies that cache-sensitive applications are granted more private space. This fact allows for improving the overall system performance (quantified with the turnaround time) and the unfairness compared to Linux default OS. Unlike existing approaches on non-inclusive caches, Cache-Poll is designed to work for a high number of cache-insensitive and sensitive applications (i.e., as many applications as the number of cores).

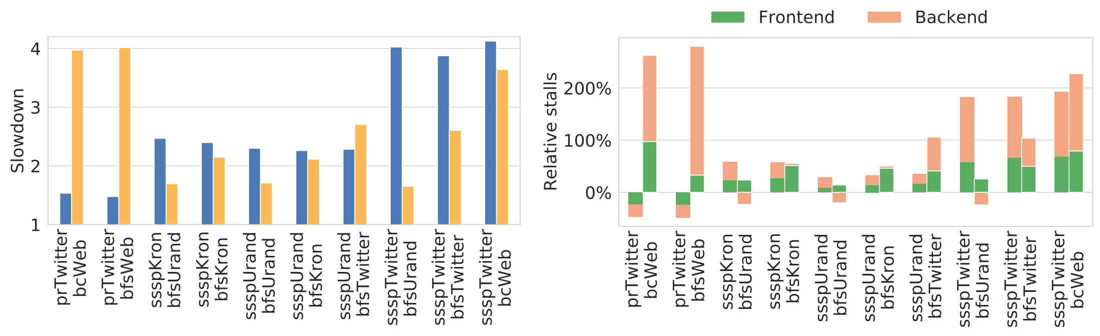
CHAPTER 6

Core Resource Management

Graph computation has gained increasing popularity as it is used for problem-solving in many contemporary domains such as social networking, big data, and machine learning [143, 144, 145, 146, 147]. The increasing computational power of current commodity multi-core processors allows processing massive graphs in a single machine [148, 149, 150, 151, 152]. Considering the scalability issues of graph applications, [66, 20, 21, 22], executing multiple graph applications concurrently is the most efficient approach to provide full system utilization. However, when multiple graph applications are scheduled together, by default, the parallel runtime creates as many threads as logical cores for each application. Then, the OS scheduler applies a time-sharing policy, which can severely damage the performance.

To overcome time-sharing weaknesses, each application should be assigned to a fraction of the processor cores, removing the intra-core interference. To this end, two main issues must be overcome: i) how many cores should be allocated to each application at each point in time, and ii) how can an application change the number of threads spawned dynamically? Previous works [64, 65, 66, 60, 63, 22, 62] have proposed spatial scheduling approaches to improve the efficiency of the execution of parallel applications by tuning the thread-level parallelism (TLP). Only the approaches proposed by Moori et al. and Srikanthan et al. [66, 65] consider graph applications. Moori et al. [66] propose a static approach that decides the TLP before the application execution starts; thus, this is not applicable in dynamic environments. In contrast, MAPPER [65] regulates application parallelism at run-time, but, as experimental results will show, it does not perform well with graph workloads.

This chapter proposes AFAIR, a fair spatial core-allocation approach to accelerate graph workloads. For a given workload mix, AFAIR dynamically determines the best CPU configuration (number of cores assigned to each application) by balancing the consumption of the shared memory resources. By assigning the optimal number of cores to each application, AFAIR minimizes the total execution time and, thus, maximizes system performance and fairness.



(a) Slowdown over individual execution **(b)** Relative stalls (%) breakdown

Figure 6.1: Effect of time sharing in 2-application workload mixes with respect to standalone execution using all cores.

6.1 Motivation: Weaknesses of the Default Linux Time-Sharing Scheduler with Graph Workloads

The standard Linux time-sharing scheduler, namely SCHED_OTHER[1], is intended to be used when no special real-time mechanisms are required and aims to ensure fair progress among all SCHED_OTHER threads [153].

When executing graph applications under time sharing, SCHED_OTHER first allocates all processor cores to one of the applications, which monopolizes the processor cores for a share of time; after that, the scheduler alternates the application by switching in another application, in a round-robin fashion. Each share of time is around 15ms in our experimental platform (see Section 6.2 for further details) when running workloads made up of two graph applications.

As a consequence, the inter-application interference rises each time the application switches out and then in again, as the incoming application replaces instructions and data of the outgoing application from the processor components (e.g., L1 data cache, L2 cache, and L3 cache). This is especially critical, as experimental results will show when executing graph applications.

We evaluated the performance of SCHED_OTHER while running workloads made up of two graph applications to check how fairly it behaves. To this end, we measured the slowdown that each application of the pair suffers with respect to individual execution when using all available cores (see Section 6.2 for full experimental details). For illustrative purposes, Figure 6.1 presents the results for ten experiments of 2-application workloads. It can be observed that, as expected, all applications experience slowdown (Figure 6.1a). We experimentally checked that the share of time is roughly the same across the running applications; however, while the performance of some applications in some workloads is only slightly affected (e.g., 1.5× slowdown), the performance of others suffers significantly, as high as 4×. Different cases can be appreciated: sometimes only one application

[1]`https://man7.org/linux/man-pages/man7/sched.7.html`

is significantly affected and the other only a little (e.g., *prTwitter* and *bcWeb*), but there is also the case where both can be severely slowed down (e.g., *ssspTwitter* and *bcWeb*).

To provide further insights on this issue, we looked into performance counters measuring processor stalls due to specific processor components at execution time. More precisely, those that account for execution stalls caused by resources of both the processor *frontend* (e.g., L1 Icache) and *backend* (e.g., D-TLB, L1 Dcache, L2 cache, L3 cache, and main memory) [140]. Figure 6.1b shows the relative increase (in percentage) of the stall cycles measured under SCHED_OTHER in the workloads over standalone execution. It can be seen that applications suffering a high slowdown are those experiencing the higher increase in the number of processor stalls. For instance, in the first workload (*prTwitter* and *bcWeb*), *bcWeb* experiences around 3.5× more processor stalls than when executed alone. This increase mainly arises in the backend, since time sharing makes applications compete for data-cache space and main-memory resources. That is, each time a new application is switched in, it replaces data from the outgoing application. Subsequent accesses to this data will miss in the cache, which will translate into a higher number of stalls. We found that most of these stalls arise due to an increase in the number of L3 cache misses. In addition, we also found the L3 miss penalty is longer under time sharing meaning that main-memory contention increases.

In the same workload, however, we surprisingly found that the number of stalls in *prTwitter* decreases over standalone execution. The reason is that the miss penalty of all the cache levels reduces. This is mainly due to *prTwitter* experiencing a high number of cache misses, and some of them are handled by the cache and memory controllers while this application is switched out. Moreover, we found that the number of misses in *prTwitter* in any cache level is by an order of magnitude greater than that of *bcWeb*.

To take away: SCHED_OTHER allocates applications in a round-robin fashion, trying to be 'fair' by allowing each thread to run the same amount of time. However, the particular conditions of graph applications—they spawn across all the processor cores and mostly have high L3 cache miss rates—makes this policy introduce important unfairness from a performance perspective. *While the performance of some applications is hardly affected (e.g., 1.5× slowdown), the performance of others can be significantly affected (e.g., by 4× slowdown), resulting in low system fairness.*

6.2 Experimental Setup

Hardware. The experiments have been conducted in the Intel Skylake processor described in Section 3.1 with the software manager we developed, described in Section 3.3. The OpenMP [154] libgomp library modifications have been performed on top of GCC version 7.5.0.

Workloads. In this work, we have used graph applications from the GAP Benchmark Suite [18] (more details can be found in Section 3.4).

Methodology. The methodology adopted to execute the workloads is as follows: applications that make up the workload are launched so that they complete the initial graph loading and building phase, which is performed sequentially. Once all applications have completed this initial sequential phase, each workload runs until all applications of the mix execute the same number of instructions (N) as they would execute if they ran alone for 1,000 seconds in 1 CPU (2 threads). In this way, we only consider the time when all applications are performing parallel computation, allowing us to assess the *fairness* of the equal-priority co-running applications based on their parallel efficiency (i.e., scalability with respect to 1 CPU). To obtain N, we increased the *number of trials* each graph algorithm runs, a configurable parameter of the GAP Benchmark Suite. The applications that do not finish last continue to run until the other applications finish. However, only the results of the first N instructions are considered.

6.3 Graph Applications' Characterization

This section characterizes the performance scalability of graph applications as the core count used to run each application increases. We also analyze the main characteristics from a resource consumption perspective that these applications exhibit at run-time.

The aim of this study is twofold. On the one hand, to provide insights regarding the best distribution of number of cores among applications. On the other hand, to develop guidelines that allow the scheduler to identify these applications at run-time.

6.3.1. Performance Scalability to Core Count

To help study scalability in performance with increasing numbers of cores, we define the **scalability factor** as the ratio of the execution time of the application executed with all the cores of the system (12 in our platform) to its execution time with only one core. In other words, this term defines the maximum performance achievable in the processor over one core using both threads.

Figure 6.2 presents the scalability trend as more cores are added to the studied graph applications from 1 up to 12, corresponding to monopolizing all processor cores. As it can be observed, most applications present a linear trend, meaning that they manage to achieve a near-perfect scalability (i.e., close to 12). Other researchers presented similar observations [77, 76] despite reporting a less linear scalability trend.

The rightmost point of the straight line defines the scalability factor. According to this value, applications can be grouped in three *scalability groups*:

- *Low scalability*. This group includes applications with a scalability factor less than 1/2 the number of cores (i.e., 6). This group contains only *bfsUrand*, which shows a scalability factor of around 4.6.

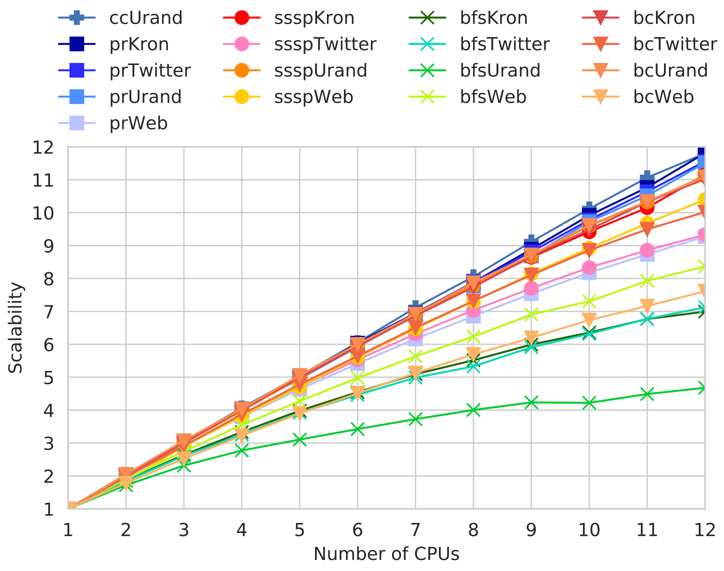


Figure 6.2: Scalability trend (w.r.t. 1 CPU) of the studied graph applications with increasing number of cores. Each core executes two threads (Hyper-Threading enabled). The scalability at 12 CPUs defines the scalability factor.

- *Medium scalability*. This category includes applications whose scalability factor is between 1/2 and 3/4 the number of cores (i.e., between 6 and 9). Applications that fall in this category are *bfsKron*, *bfsTwitter*, *bfsWeb*, and *bcWeb*.
- *High scalability*. Includes applications whose scalability factor is over 3/4 the number of cores (i.e., over 9). The remaining applications fall into this category.

The fact that there are applications showing different scalability trends means that when running multiple graph applications together, those applications that present a lower scalability factor (i.e., medium and low scalability) will require a higher number of cores than those that present high scalability, to balance the applications' progress and provide system fairness.

To take away: We observe that, among the studied graph applications, most applications show high scalability to core count in contrast to a small proportion of applications that show medium or low scalability. If the scheduling policy is aimed at maximizing system utilization considering fairness, when running multiple graph applications that show different scalability trends, *it should favor the application(s) showing the lower scalability factor by assigning it more cores so it can spawn more threads.*

6.3.2. Identifying Applications' Scalability

As mentioned above, the scheduler should be capable of discerning at execution time those applications showing high scalability from those experiencing lower scalability, in order to decide how many cores should be allocated to each application.

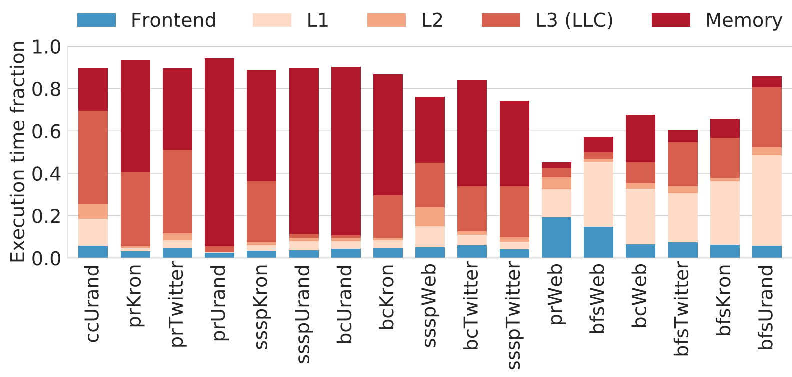


Figure 6.3: Processor stalls (fraction) breakdown over the total execution time: frontend and memory hierarchy (L1, L2, L3 caches and main memory).

For this purpose, and in order to provide insights about how to identify the scalability group each application belongs to, we looked into the major system resources that can potentially be a bottleneck to application performance. More precisely, we have quantified the number of processor stalls and classified them according to the processor side where they come from (i.e., frontend or backend). Figure 6.3 shows the fraction of stalls [140] (averaged over the total execution time) across all threads of the studied graph applications when they execute in all the processor cores. Frontend stalls, mainly due to the instruction cache and instruction TLB, are presented together as they represent a small fraction of time. Backend stalls can be split into core and memory stalls, but the plot only presents memory stalls classified according to the memory structure that causes the stall, as core stalls were close to zero in our platform. Applications are sorted in descending order of scalability factor (see Section 6.3.1), *ccUrand* being the application showing highest scalability (11.9) and *bfsUrand* the application with lowest scalability (4.6).

It can be appreciated that the 11 most scalable graph applications (i.e., the 11 leftmost applications) show that, for a high fraction of their execution time (over 60%), the processor is blocked due to backend stalls. This corroborates the observation of previous works [76, 77, 78]. The main contributor to stalls is main memory, although the L3 cache is also equally significant or even higher in some applications (e.g., *ccUrand*). The only exception is *prWeb*, whose execution time is not dominated by backend stalls.

To provide insights into the reasons why some applications scale poorly, we looked into the graph algorithms and input graphs. We found that the `bfs` algorithm presents an efficient code that needs to check fewer vertices [76] so both the L3 cache and main memory are less critical resources. We also found that the Web graph contains significant spatial locality as it leads to longer contiguous accesses to the L1 data cache [78]; in addition, the Web graph is the smallest one so it experiences fewer L3 cache misses.

Regarding the remaining applications (*bfsWeb* to *bfsUrand*), which are grouped as medium and low scalability, they show several important differences with respect to highly scalable applications. First, these applications show lower sensitivity to the backend (i.e., the fraction of time is lower), except for *bfsUrand*. Among the backend stalls, most of these are due to the L1 data cache. These stalls are mainly caused by or at the L1 D-Cache (e.g., a delay hit waiting for an in-flight load or stalls due to store-to-load forwarding). Notice that these stalls are assigned to the L1 cache and not to a lower cache level, as they are not caused by an L1 load miss pending. Therefore, applications with medium and low scalability are less sensitive to interference at the memory resources shared among cores (i.e., LLC and main memory), since most of their performance is limited by the L1 data cache (private to each core). A particular case occurs with the poorly scalable application *bfsUrand*, which presents a significant fraction of stalls due to the LLC. We observed that, unlike the other studied applications, it presents a high hit rate in the LLC.

Independently of the application's scalability, the L2 cache has little effect on the execution time since graph applications show a reuse distance larger than that serviced by the L2 cache [71].

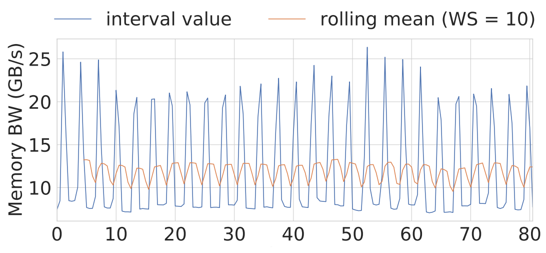
To take away: From the previous analysis, three main conclusions can be drawn:

- Applications that are highly limited (over 0.50) by one shared-memory component (e.g., the L3 cache or main memory) show the highest scalability.
- Applications showing medium to low scalability are limited by memory but to a lesser extent. On the contrary, the L1 cache has a significant impact.
- The application showing the lowest scalability, *bfsUrand*, is also limited by the LLC but, unlike the other applications, it shows a high hit rate in this cache level.

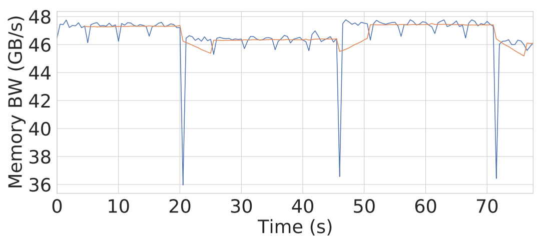
6.3.3. Parallel Regions Dynamic Memory Behavior

The GAP Benchmark Suite is comprised of a set of graph algorithms that perform parallel computation using OpenMP. Among the studied graph applications, some, like *ssspKron*, present a single parallel region that lasts the entire execution, but most applications include multiple parallel regions with different characteristics. The length of the parallel regions varies from less than a second to over 20s.

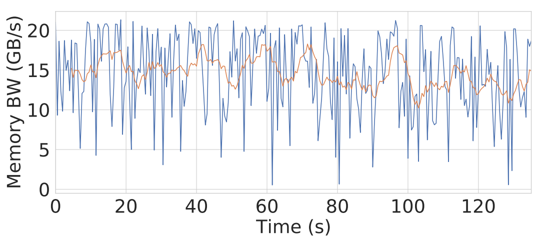
Having different types of parallel region means that an application can vary its needs from a resource consumption perspective depending on the parallel region it is executing. To confirm this, we checked for possible execution phases from a main memory perspective, as this is the processor resource that most limits performance. We found that seven applications (*bcUrand*, *ccUrand*, *prUrand* and the four graphs running with *sssp*) present a regular consumption across their execution despite consisting of multiple parallel regions. In contrast, in the remaining ten applications, the bandwidth consumption changes from one parallel region to another. To illustrate the different memory trends, Figure 6.4 shows the memory bandwidth consumption in GB/s, both raw values per second (interval value) and the rolling mean (window size of 10 seconds) of three applications across their execution when launched alone in the experimental platform with all cores available. Both *ccUrand* and *prKron* present different, well-differentiated execution phases. However, if we compare the rolling mean values, the average consumption of *ccUrand* is roughly the same across the different phases whereas *prKron* varies its consumption in each one, presenting an irregular bandwidth consumption. Another



(a) *ccUrand* (regular BW consumption)



(b) *prKron* (irregular BW consumption)



(c) *bfsTwitter* (irregular BW consumption)

Figure 6.4: Main memory bandwidth consumption over time.

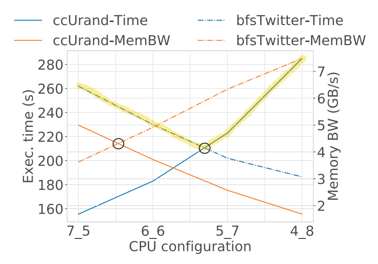
example of an irregular behavior is shown in Figure 6.4c. In this case, *bfsTwitter* does not present such well-defined execution phases like *ccUrand* and *prKron* but varies its consumption in a more bursty behavior.

The irregular behavior of some graph applications, together with the fact that obtaining the number of cores that an application should be assigned with offline profiling is too costly, motivated us to explore the possibility of enabling applications to adapt their number of spawned threads at execution time based on the target number of cores assigned by the scheduler.

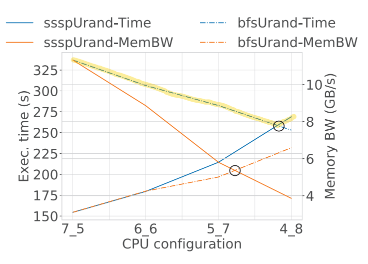
6.4 Graph Workloads Spatial Scheduler

Time sharing behaves *unfairly* from a performance point of view, because applications' performance is differently affected despite having the same share of time, when a fraction of their data blocks are evicted by the incoming applications. A way to deal with this shortcoming is to avoid applications from sharing the cores, for instance, dividing the cores equally among the co-running applications. However, considering that graph applications present different resource demands, and thus, scalability factors, applying such an equal core distribution policy may not yield the system to its best performance.

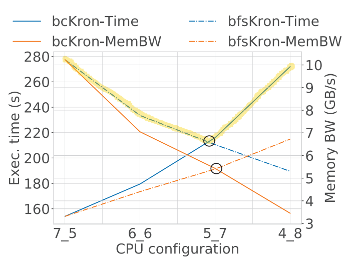
A key functionality of the scheduler is to find the best performing *CPU configuration*. A CPU configuration for an n-application workload, is defined by n variables $x1\_x2\_..\_xn$, where xi states the number of cores assigned to application i, and the sum of them equals the number of processor cores. For instance, for a workload consisting of two applications, A and B, the configuration x_y denotes that applications A and B, have assigned x and y cores, respectively, $x + y$ being the total number of processor cores.



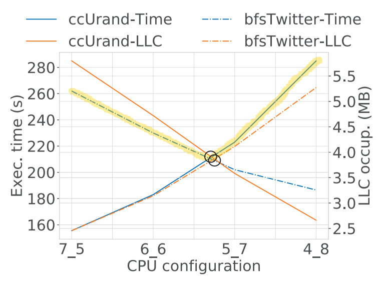
(a) Memory BW *ccUrand-bfsTwitter*



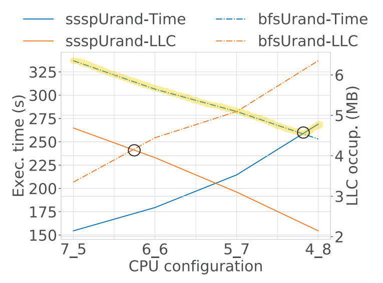
(b) Memory BW *ssspUrand-bfsUrand*



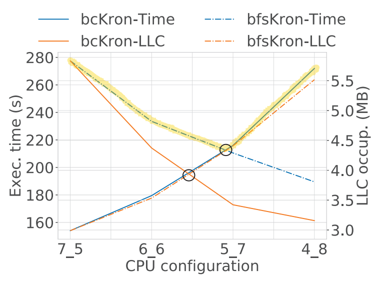
(c) Memory BW *bcKron-bfsKron*



(d) LLC occup. *ccUrand-bfsTwitter*



(e) LLC occup. *ssspUrand-bfsUrand*



(f) LLC occup. *bcKron-bfsKron*

Figure 6.5: Relationship between the execution time and the metrics memory BW (top row) and LLC occupancy (bottom row) for different CPU configurations. The point of interests are marked with a black circle. Turnaround time is highlighted in yellow.

To quantify co-location performance, we consider the turnaround time. A possible approach to find the CPU configuration that results in the best turnaround time could be using a thorough offline profiling analysis. This process, however, is generally complex and costly (even more so with increasing numbers of co-running applications) [155]. Thus, it becomes impractical in a real environment where many *unknown* applications are constantly encountered [156].

To address this issue, the proposed scheduler estimates dynamically at small phases of execution time (e.g., several OS quanta) the best distribution of cores among applications. To this end, the applications must have the capability of adapting their spawned threads dynamically at run-time depending on the number of cores assigned by the scheduler, which is challenging. Below, we discuss how AFAIR addresses this challenge.

6.4.1. Determining the Best CPU Configuration

Deciding the number of cores each of the co-running graph applications should be assigned to minimize turnaround time is challenging, especially when there is no information regarding the work each application must complete. Let's clarify this challenge through an example. Assume applications A and B are executed at some point under CPU configuration x_y. The challenge lies in finding out at execution time if releasing one core from a given application to give it to the other (e.g., change the CPU configuration to $x - 1\_y + 1$) would improve the turnaround time.

To deal with this challenge, AFAIR only makes use of online information gathered at each execution quantum from specific hardware counters. In this regard, we analyzed the relationship between the simultaneous execution of multiple (e.g., two or three) graph applications varying the CPU configuration (i.e., the number of cores assigned to each application). We mainly collected metrics that quantify the performance of major components of the memory hierarchy, since the performance of these applications is mainly affected by these components (see Section 6.3.2).

The LLC and the main memory are the two major resources showing the highest contention (i.e., source of performance bottleneck), both in standalone and in multi-program execution. Figure 6.5 plots the relationship between the execution time varying the CPU configuration and the studied metrics (main memory bandwidth, top row, and LLC occupancy, bottom row of plots) across three pairs of applications presenting three representative behaviors. Pairs of applications are made up of a highly scalable application (*ccUrand*, *ssspUrand* or *bcKron*) and a medium (*bfsTwitter*, *bfsKron*) or a low scalable (*bfsUrand*) application. Every pair has been executed under four CPU configurations x_y as they are the most impacting ones on performance, where x and y are the number of cores assigned to the application represented by solid lines and dashed lines, respectively. Notice that $x + y$ equals 12 across the four configurations. Each plot shows two pairs of lines: the blue lines show the trend of the execution time for each application, and the orange lines show the trend for the studied metric. The point that minimizes the turnaround time is that when both blue lines meet (marked by a black circle). The turnaround time trend of the workload, highlighted in yellow, is defined by the slowest application at any point of time. Notice that if the orange lines meet around the same point (also marked by a black circle) as the blue lines, then this metric can be considered as a good proxy for turnaround time.

Two interesting observations can be made. Firstly, the best CPU configuration varies depending on the scalability group of the less scalable application. The workloads that contain a medium scalable application experience the best turnaround time in configuration 5_7 while the workload containing the less scalable application (*bfsUrand*) reaches such a point beyond the configuration 5_7 towards 4_8. Results present the values averaged across all the quanta of the execution. Some applications present execution phases that may show different trends (see Section 6.4.2), therefore, the configuration resulting in the best turnaround time can dynamically vary according to the execution phase (see Section 6.5.4 for further details).

Secondly, none of the metrics performs necessarily better than the other but it depends of the workload. LLC occupancy performs best in workload *ccUrand-bfsTwitter* as *ccUrand* is limited by LLC space (0.44 L3 cache compared to 0.21 main memory in Figure 6.3). For the other two workloads, memory bandwidth is the best performance indicator since *ssspUrand* and *bcKron* introduce a higher pressure on the memory bandwidth (0.77 and 0.58, respectively), compared to the LLC (0.02 and 0.19, respectively). A special case can be observed for *ssspUrand-bfsUrand* (Figure 6.5b), where the orange lines (memory bandwidth) meet a bit before the point where the blue lines (execution time) meet. As mentioned in Section 6.3.2, *bfsUrand* presents a high hit ratio in the LLC (over 0.8); therefore, the scheduler could enhance the performance of this application by providing it with more cores so it has access to a larger share of the resources monitored (i.e., LLC and memory bandwidth) as the application generates more threads.

To take away: This analysis has illustrated the approach AFAIR follows to *estimate the CPU configuration that yields the lowest turnaround time by identifying and* balancing *the consumption of the shared memory resources (LLC or main memory) that most harms the performance of the highly scalable applications.* Notice that this happens as a side effect of spawning the applications in more threads; however, no specific technology (e.g., Intel CAT [10]) is used to assign more cache space to the target application. In the case of a low scalable application, the balance point moves to favor (i.e., more cores should be assigned to) such an application.

6.4.2. Modifying the Number of Spawned Threads Dynamically

The GAP Benchmark Suite applications are parallelized with Open-MP. By default, OpenMP checks the number of cores and spawns the application in as many threads as logical cores it detects only at the start of the execution. We modified the GNU OpenMP library (libgomp) so that the CPU affinity is checked before the start of each parallel region, and the number of threads is tuned based on the number of allocated CPUs. To obtain the number of CPUs assigned to the running application, we have implemented the function `GetCPUCount()`, which uses the system function `sched_getaffini- ty` to find the number of currently assigned CPUs.

To use AFAIR's extension, the path of the modified libgomp library must be added to `LD_LIBRARY_PATH` environment variable. This implementation allows AFAIR to run transparently without needing to modify the source code of OpenMP applications. Our extension could be easily applied to other parallel runtimes as it requires changing and adding a few code lines.

6.4.3. Putting it All Together: AFAIR

This section introduces the AFAIR approach, which applies performance criteria to dynamically adapt the affinity of graph applications (and thus, the number of spawned threads) to CPUs with the aim of minimizing the turnaround time of co-located applications, providing system fairness. AFAIR consists of five main steps discussed below.

Initial CPU configuration.

When cores are not shared in a time-interleaving fashion, the execution time of workloads is significantly reduced. The number of processor stalls are decreased by i) removing the intra-core interference, and ii) reducing the off-core interference as processor stalls reduce when requests from multiple applications are allowed to compete among them, at the same time, for LLC and main memory. Therefore, since AFAIR has no information about the co-running applications in the first quantum, it assigns the same number of cores to each application (e.g., 6 cores in case two applications are scheduled).

Application behavior profiling

To determine the behavior each application presents, we sample and monitor performance counters for a period of Q quanta. At the end of each profiling period, the metrics of interest are computed (stalls due to L1 cache, LLC, and main memory; LLC occupancy and memory bandwidth), which are used to classify applications' behavior. This process is performed in two steps:

(i) **Determining the scalability (high/low) trend of each application.** First, we identify the scalability trend each application presents by looking at the peak L1 stalls (95% percentile) values. Those applications in which L1 stalls represent a significant fraction of the total execution time[2] are classified as poorly (i.e., medium to low) scalable applications. Otherwise, they are considered as highly scalable applications.

(ii) **Determining the memory performance bottleneck.** As shown in Section 6.3.2, scalable applications' performance bottleneck is mainly due to the LLC or main memory. Therefore, the metric of interest is defined depending on the resource that harms most the performance of highly scalable applications: the LLC occupancy for applications whose performance is limited by the LLC, and the memory bandwidth for applications with dominant stalls introduced by main memory.

Testing if a new core configuration needs to be applied

Regardless of the metric used as a proxy, we found that the best configuration to minimize the turnaround time is located at the point where the metric of interest is balanced for both applications (see Figure 6.5). Therefore, AFAIR continues releasing cores to the application with the highest number of stalls introduced by the L1 cache until reaching this point. However, if the application scaling worse presents a high hit rate in the LLC, this means it presents low scalability, and thus, the balance point is relaxed so that this application receives a slightly higher share of the resource.

In case there are three applications running simultaneously, if there is more than one highly scalable application, the one chosen as a candidate to release a core is the application showing the highest metric value. Likewise, among the applications with medium or low scalability, the one chosen to receive the released core is the application presenting the lowest metric value. Notice that when running three applications, the application that releases the core is not always the same, as its behavior is likely to change with one core less.

[2]E.g., we found that over 0.3 presents the best results in our experimental platform.

Applying the new core configuration

If the criteria is met to change the core configuration (i.e., transfer one core from one application to another), it is not applied immediately in the next quantum. Instead, as explained previously, the approach needs to wait until applications finish their current parallel region before being able to change the number of spawned threads to each co-running application.

Applications, however, do not finish their parallel regions at the same time but they run in a non-synchronized way. Therefore, the final core balancing cannot be applied until that point of time. To deal with this fact, the application that finishes earlier continues running with the *new* number of cores estimated by the approach. Meanwhile, the other application remains running with its actual number of cores. Therefore, two scenarios can occur: i) a new core is allocated to the application that finishes earlier and used in time sharing by both applications, and ii) a core is released and not used by any of the applications. Nonetheless, the number of quanta this situations occur are very small compared to the total execution quanta (see Section 6.5.5).

Due to the constraint explained above regarding the response time of applying a new configuration, a bad decision could lead to important performance losses. Therefore, a new configuration is applied only if the *tendency* changes, i.e., the observed behavior is maintained throughout two or more consecutive phases.

Roll back to a previous core configuration

Applications may or may not experience regular behavior during the whole execution (see Figure 6.4). Thus, there may be execution phases where the highly scalable application requires the core it had previously released in order to sustain its performance. This can be detected if the highly scalable application presenting the lowest metric value (e.g., lowest memory bandwidth) drops below a certain level (e.g., 60% of the fair proportion). In such a case, the application with low scalability that presents the highest metric value returns a core to this application.

6.5 Performance Evaluation

6.5.1. Workload Mixes

To evaluate the scheduling approaches, we have generated 40 2-application workloads and 25 3-application workloads. Due to the fact that graph applications have huge memory requirements (tens of Gigabytes) and our experimental platform has a relatively small number of cores, workloads with a higher number of graph applications cannot be run.

To test the ability of AFAIR to adapt the cores' allocation, we have mixed applications belonging to different scalability groups. Table 6.1 shows the number of applications of

# of apps.	Workloads Groups	# Apps. grouped by scalability		
		Low	Medium	High
2	#1 to #12	1	0	1
	#13 to #40	0	1	1
3	#1 to #5	1	1	1
	#6 to #15	1	0	2
	#16 to #18	0	2	1
	#19 to #25	0	1	2

Table 6.1: Characteristics of the workloads generated.

each scalability group contained in each workload. Notice that when running applications with similar scalability factor (i.e., they belong to the same scalability group), the best CPU configuration is the same as that applied by Equal, which is the starting point of AFAIR. The workloads within each group contain the same amount of applications showing high, medium or low scalability and have been sorted in descending order of turnaround time gain obtained with AFAIR with respect to Equal.

6.5.2. Compared Schedulers

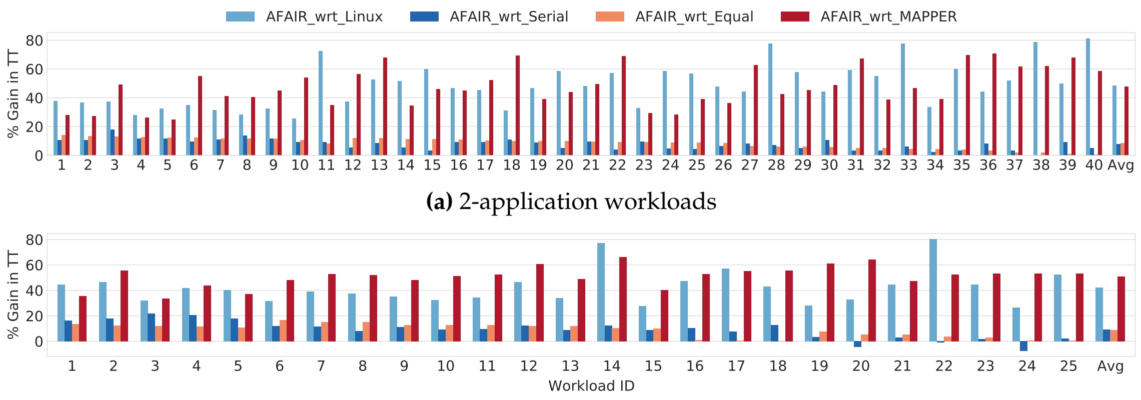
AFAIR is compared with the following scheduling policies:

- **Linux.** Applications are executed with Linux's default scheduler, which applies a time-sharing scheduler. Each application launches as many threads as possible (equal to the total number of cores).
- **Serial.** Applications are executed serially, one after another, so that they execute alone with all possible cores.
- **Equal.** Applications are executed with the same number cores, each launching only as many threads as assigned cores.
- **MAPPER.** State-of-the-art approach proposed by Srikanthan et al. [65], which controls the degree of parallelism of each application and the specific cores allocated.

6.5.3. Experimental Results

This section evaluates the performance of AFAIR, both in terms of system utilization (quantified with the turnaround time or TT) and system fairness. For each workload, results for all the studied policies are shown.

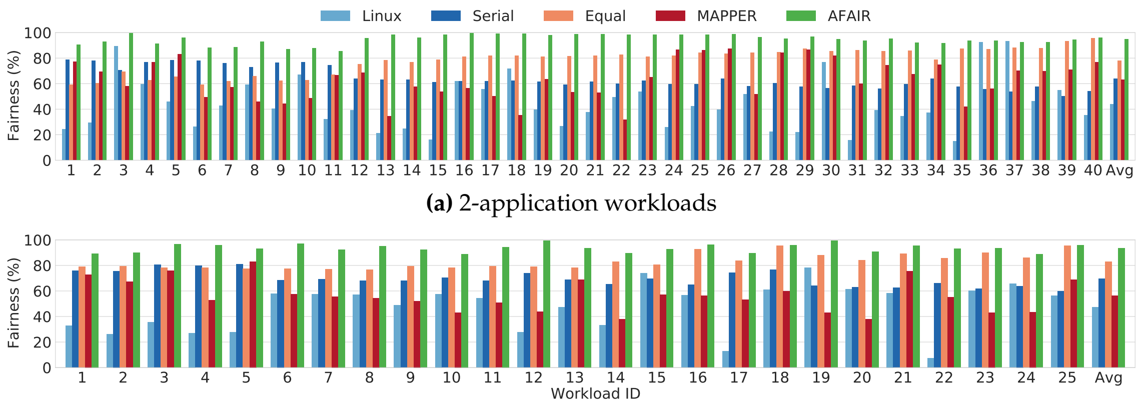
Figure 6.6 presents the performance gains (in %) in TT AFAIR achieves over the studied scheduling policies across the studied 2- and 3-application workloads. As observed, regardless of the number of co-running applications, AFAIR clearly outperforms Linux and MAPPER by 45% and 49% on average, respectively. Similar to Linux, MAPPER allocates CPUs at logical-core level, meaning that two different applications can share the same



(a) 2-application workloads

(b) 3-application workloads

Figure 6.6: Gain (in %) of the turnaround time (TT) obtained with AFAIR with respect to the other evaluated scheduling policies.



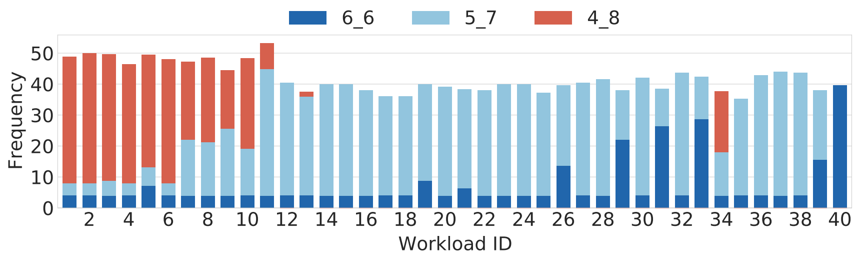
(a) 2-application workloads

(b) 3-application workloads

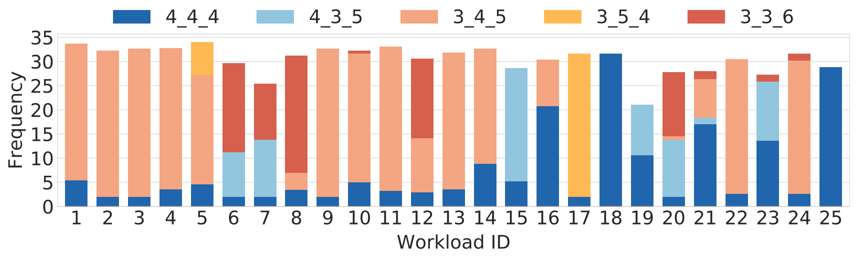
Figure 6.7: Fairness results (in %) obtained with all the the evaluated scheduling policies.

physical core. Unlike other scientific compute-intensive benchmarks, graph applications are memory-bound and experience important performance constraints (e.g., caches and TLB misses) specific to the cache hierarchy and main memory. Therefore, sharing the same core can yield the system to severe performance degradation due to the huge intra-core interference at the D-Cache and D-TLB (see Figure 6.3). For instance, this is the case of workload #28 composed of *ssspWeb* and *bfsKron* applications. Consequently, and due to sharing the core introducing highly unpredictable behavior, Linux and MAPPER policies work poorly in fairness (see Figure 6.7), which is by 44% and 62% in 2-application workloads, respectively. Similar results can be observed with 3-application workloads.

Compared to Serial and Equal policies, where applications are not allowed to share the core, AFAIR improvements reduce, but even so, they are on average by 8%, and rise up to 18% in some applications. Moreover, as shown in Figure 6.7, these performance gains are achieved while significantly improving fairness. While Serial and Equal achieve an average fairness by 62% and 78% in 2-application workloads, AFAIR obtains almost perfect fairness above 98% in nearly half of the workloads giving an average fairness by



(a) 2-application workloads



(b) 3-application workloads

Figure 6.8: Frequency each CPU configuration has been applied in the studied workloads.

95%. Notice that in workloads #38 to #40, AFAIR matches the performance achieved by Equal. The reason is that these workloads reach their best TT when assigning half of the cores to each application. Results slightly differ in 3-application workloads.

6.5.4. Analyzing AFAIR's Dynamic Behavior

To demonstrate the effectiveness of AFAIR in adapting the CPU configuration at run-time, Figure 6.8 shows the number of times each CPU configuration x_y or x_y_z has been applied in each of the studied 2-application and 3-application workloads, respectively. To ease the analysis, the relationship between applications and the corresponding number of cores in the CPU configuration is always from highest to lowest scalability group. This means that, for a given workload, x always corresponds to the number of cores assigned to the highly scalable application, and the last value (y in x_y and z in x_y_z) always corresponds to the number of cores assigned to an application showing medium or low scalability. For 3-application workloads, y can be an application of any scalability group.

For most workloads, the configuration that allocates equal number of cores (6_6 in Figure 6.8a and 4_4_4 in Figure 6.8b), which is the starting configuration in AFAIR, is only applied for one or a low number of phases. For example, workload #5, which corresponds to pair *ssspUrand-bfsUrand* in Figure 6.5, manages to converge quickly to configuration 4_8, achieving performance gains of 12.5%. This means that AFAIR is able to rapidly detect the core-allocation demand needs of applications.

Among those workloads that achieve lowest performance gains with respect to Equal (see Figure 6.6), most of the execution phases have the default Equal configuration, while those that obtain highest performance gains have only one or a few phases with Equal.

Nonetheless, some applications benefit from a longer time in 6_6, like workload #26 (Figure 6.8a), which corresponds to pair *ccUrand-bfsTwitter* in Figure 6.5. Despite the fact that the configuration that achieves the lowest turnaround time on average is 5_7, in 35% of the execution phases, the configuration applied is 6_6. Recall that *bfsTwitter* presents an irregular memory bandwidth consumption (see Figure 6.4c).

In general, AFAIR assigns more cores to applications that present low scalability. For example, in the 2-application workloads, AFAIR allocates 8 cores in most of the phases to the applications that present low scalability and 4 cores to its highly scalable co-running application (workloads #1 to #7 in Figure 6.8a). Additionally, results show that having multiple configurations also brings significant gains. For instance, in the 2-application workloads #8 to #11, performance gains rise by 11%. In these workloads, AFAIR applies the configuration 5_7 in a similar number of phases as configuration 4_8. As explained in Section 6.4.1, applications may have different needs in different execution phases. Another example can be observed in workload #6 in Figure 6.8b, where AFAIR adopts in 62% of the phases the configuration 3_3_6 and in 31% of the phases the configuration 4_3_5. This proves that a dynamic core-allocation policy can bring substantial performance gains.

6.5.5. Overhead of AFAIR

The overhead of applying AFAIR comes mainly from reading the performance counters with Linux Perf and measuring the memory bandwidth and LLC occupancy with Intel RDT, which is minimal. As explained in Section 6.4.3, when the CPU configuration is changed, it is not applied immediately in the next quantum. All executing applications must finish their current parallel region before changing the number of spawned threads. Thus, there can be some quanta where some cores are *time shared* or not used at all (see Section 6.4.3). Nonetheless, we found that these situations occur, on average, only in 2% and 5% of the execution quanta in the 2- and 3-application workloads, respectively.

6.6 Summary

Current data centers seek to maximize the use of hardware resources. Among the workloads of interest, graph computation has become essential today to solve problems in many application domains, such as scientific computing, social networks, and big-data analytics. Thus, optimizing the execution of graph applications is of high interest.

This chapter has presented AFAIR, a disruptive core-allocation policy that launches multiple graph applications simultaneously, allocates each a subset of cores, and adapts the number of threads of the application to be launched dynamically to maximize both system utilization and fairness. Moreover, the number of threads spawned by the running applications is changed at run-time without any modification in the applications' source code.

Experimental results show that AFAIR manages to obtain near-optimal system fairness. At the same time, it improves system utilization by reducing the total turnaround time significantly. AFAIR overcomes the weakness of the default Linux time-sharing policy and improves the performance of other scheduling policies commonly used (Serial and Equal), as well as outperforming the state-of-the-art approach MAPPER.

Part II

Cloud Computing

CHAPTER 7

Workload Characterization in the Public Cloud

Cloud computing, compared to HPC, poses new challenges to computer architects. On the one hand, cloud systems deploy a virtualized environment and are composed of multiple components that interact with each other, including server nodes, client nodes, storage nodes, and interconnection networks. Unfortunately, existing studies [54, 84, 50, 31] omit important system components or do not consider the appropriate software environment, resulting in a loss of representativeness. On the other hand, among cloud applications, latency-critical applications deserve special interest as the performance of these applications is defined by the *tail latency* (typically 95th or 99th percentile) since QoS violations cannot be overlooked but could be hidden under the average latency metric.

This chapter characterizes a set of representative multithreaded latency-critical applications (Tailbench [125], and `media-streaming` [19] applications) in a production-like environment, implementing the full system stack (hardware and software). First, we study the impact of varying the number of threads or scalability, as well as the effect of Hyper-Threading (i.e., threads running on the same core), on the performance –tail latency and QPS while satisfying QoS– for each application. Results reveal two main counter-intuitive findings regarding the insensitivity of applications to the number of server threads (non-scalable) and Hyper-Threading. With the aim of detecting these applications at run-time, we study the correlation between these applications and the utilization of each major (CPU, LLC, main memory bandwidth, disk bandwidth, and network bandwidth) system resource showing that measuring the utilization and trend of these resources can help in the detection. Finally, by applying existing technologies, we study the impact of inter-VM interference at the main shared resources other than the CPU.

All the studies are made from two main perspectives. On the one hand, research conclusions are highlighted in small *to take away* sections. On the other hand, practical actions oriented to the cloud provider are highlighted in small *cloud provider actions* sections that include hints to improve performance and the utilization of the major resources.

7.1 Experimental Methodology

7.1.1. Latency-Critical Applications

Latency-critical applications are widely used in cloud environments. In these applications, the cloud server typically implements online services (e.g., speech recognition or language translation) and must respond to the input requests within specific latency bounds to guarantee QoS and provide a satisfactory user experience.

As a representative set of latency-critical applications, we use the TailBench benchmark suite [125] (more details in Chapter 3.4), which includes applications from diverse set domains that exhibit a wide range of tail-latency behaviors. In addition, this work also analyzes the `media-streaming` workload from CloudSuite [19] as none of the studied Tailbench applications exhibit such behavior since they all present negligible network demands.

7.1.2. Setting Representative Load Levels

Unlike HPC benchmark suites, TailBench applications include the QPS parameter that allows generating a wide range of load levels. For experimental purposes, this parameter needs to be tuned to match the desired workload level. Below, we discuss the approach followed to obtain a representative range of QPS values for each of the studied applications.

To simulate real client-server behavior, clients of TailBench applications issue requests to the server following the Zipfian[1] distribution [126, 127]. To do so, the clients use a *request generator* to indicate the points of time when requests are issued to satisfy the demanded QPS and the Zipfian distribution. Unfortunately, when a large QPS is demanded, it may happen that the client cannot generate and send the requests fast enough. In this scenario, the client might still reach the desired QPS on average but break the Zipfian distribution, which compromises the representativeness of the experiment.

To address this issue and make experimental results representative, we defined the metric *timely requests ratio*, which accounts for the percentage of requests that the clients can issue fulfilling the request times generated following the Zipfian distribution. A request is defined as *non-timely* when the request time provided by the request generator is earlier than the current time. Notice that clients can break the distribution and still meet the requested QPS on average. In this regard, we found that even though a single client can generate up to thousands of requests per second, usually, it can only generate between 400 and 600 without breaking the distribution. To provide representative results, we checked that the target QPS is achieved while guaranteeing that at least 97.5% of the requests are timely.

[1]The Zipfian distribution is a related discrete power law probability distribution that states that, for a given set of items (e.g., words of a text), the frequency of any item is inversely proportional to its rank in the frequency table.

Workload	QPS range	Number of clients
`img-dnn`	100 – 3100	12
`masstree`	250 – 2500	12
`moses`	10 – 1100	6
`shore`	10 – 2000	12
`silo`	250 – 4000	18
`specjbb`	250 – 3500	18
`sphinx`	0.2 – 8.5	2
`xapian`	100 – 2800	4

Table 7.1: QPS range and number of clients used for each workload to guarantee that the ratio of timely requests is above 97.5%.

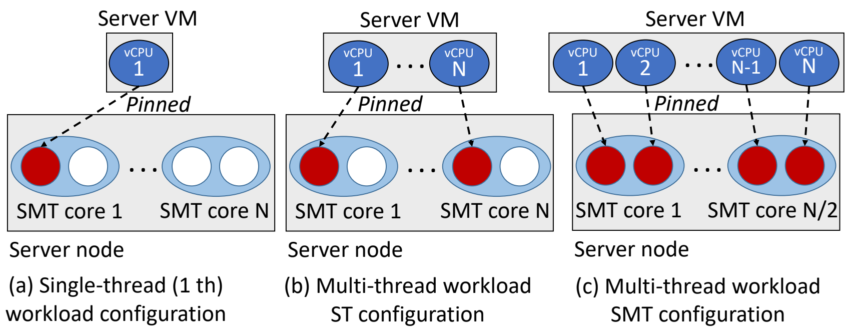


Figure 7.1: Sever VM core configurations studied. N stands for the number of threads spawned by the server, each assigned to a different vCPU.

Under this condition, we explored the QPS range of each workload as well as the required number of clients to generate the requests. QPS steps were chosen to cover a wide range of representative CPU loads, from 10% to saturation (over 50%). This allows studying the behavior at a usual average utilization [157] (e.g., 20%) and how it is affected as the load grows. Table 7.1 presents the results. As observed, QPS widely varies among applications, ranging from 0.2 to 8.5 in `sphinx` and from 250 to 4000 in `silo`.

Unlike TailBench applications, the load of `media-streaming` is indicated as the number of *sessions*. Any number of clients can be launched to complete the target number of sessions. In this work, we explore the client load up to a maximum of 70 sessions and 24 clients since we found out that a higher number of sessions yields saturated results.

7.1.3. Studied Server Scenarios

We characterize the behavior of Tailbench applications and *media-streaming* across different server configurations or scenarios. Scenarios have been designed to evaluate both the effect of Hyper-Threading and the scalability when the server application spawns a higher number of threads. We assume that each thread is assigned to an individual virtual CPU (vCPU).

Three main configurations have been considered, as shown in Figure 7.1, to generate scenarios of interest. In configuration (a), only one vCPU is used, pinned to a logical core, which is in charge of running the single-threaded server application. In configuration (b), the server application runs N threads (assigned to N vCPU cores), and each one runs in single-threaded mode in a different physical core.

Finally, configuration (c) differs from configuration (b) in that the N threads (vCPU cores) are pinned in pairs to the two logical cores of the same SMT physical core, running both threads concurrently. Notice that scenario (c) uses half the number of SMT cores of scenario (b).

Under these configurations, N has been evaluated both for two and eight threads or vCPUs, providing five main scenarios. More precisely, configuration (b) splits into scenario *2-ST* (2 threads are assigned to 2 cores working each in single task mode) and *8-ST*, and configuration (c) breaks down into *2-SMT* (2 threads are pinned to half the number of cores working in multi-task mode) and *8-SMT*. Notice that configuration (a) can be seen as a particular case of configuration (b) for N equals 1, so it will be referred to as *1-ST* and represents the fifth scenario. These scenarios allow comparing the behavior of a single-threaded server against a multithreaded server, as well as the impact of Hyper-Threading in a relatively high multithreaded (i.e., eight threads) server.

7.1.4. Studied Performance Metrics

To characterize the workloads in the above scenarios, we need to define the metrics of interest. The characterization studies presented in this work cover both performance and resource utilization Below, we list the studied metrics.

- 95^{th} **tail latency**, i.e., the 95^{th} percentile of the *sojourn* (end-to-end) times, which considers both queue and service times. This percentile indicates that only 5% of the responses take longer to complete than that value. In the case of *media-streaming*, it shows the 95^{th} percentile of the server response time. In this thesis work, we consider that this metric defines the QoS (see Section 7.2).

- **CPU utilization**. This metric refers to the average CPU utilization of the logical cores assigned to the VM (vCPU cores), e.g., for the 2-thread scenarios, we report the average utilization of the two CPUs where the application is running. To obtain the utilization of each CPU, we use the data collected from the file /proc/stat, which reports statistics about the kernel activity aggregated since the system first booted. The CPU utilization quantifies the fraction of time the application runs on the processor (i.e., percentage of active time with respect to the total time). Notice that this time also accounts for the time the application waits for main memory accesses even though the processor is stalled and no instruction can be executed.

- **LLC occupancy** and **main memory bandwidth** show the sum of the LLC capacity occupied and main memory bandwidth consumed, respectively, by the vCPU cores.

Workload	Tail latency QoS	QPS single-thread
`img-dnn`	3.6 ms	600
`masstree`	1.4 ms	1000
`moses`	7.1 ms	250
`shore`	25 ms	90
`silo`	0.5 ms	700
`specjbb`	0.7 ms	1500
`sphinx`	4275.4 ms	0.6
`xapian`	6.2 ms	300
`media-streaming`	500 ms	25

Table 7.2: Tail latency QoS (LQoS) requirements for the Tailbench workloads in our experimental platform.

- **Network bandwidth**. Two main bandwidths can be considered from the server (i.e., running-VM perspective): received and transmitted. The latter is dominant, and the former is almost negligible, so the study focuses on the latter one.
- **Disk bandwidth**. This metric refers to the disk bandwidth consumed by the running VM at the server node.

7.2 QoS and Tail Latency Analysis

7.2.1. Defining QoS

Nowadays, many online-service workloads present tail latency QoS constraints. These constraints are part of the Service Level Agreement (SLA) in some cases; in other cases, users should make sure of hiring enough resources to meet their target QoS. To determine realistic tail latency QoS constraints in our experimental setup and evaluate whether the workloads meet the QoS, we define the QoS requirement for each workload as a function of the average service time. This approach is based on the one proposed by Delimitrou *et al.* [158]. The QoS target for each workload is defined as 5$\times$ the average service time achieved with a CPU utilization of 20% in the 1-ST scenario. We will refer to this value as **LQoS**. Experimental results in this chapter assume that QoS is met whenever the 95^{th} tail latency is lower than the corresponding LQoS.

Table 7.2 presents the LQoS values (in ms) and the QPS supported by the single-threaded server for each workload that meets the QoS latency constraint. Tail latency requirements range from 0.5 ms for *silo* to 4275.4 ms for `sphinx`. Notice that the LQoS of `sphinx` is over 4 s, which may seem rather high in comparison with other speech recognition services. However, we found that these values are in line with the results of this application presented by other researchers [125]. Finally, in *media-streaming* LQoS is defined in terms of 95^{th} percentile of the response time [159]. Many web and streaming services are time-bounded, requiring the response time to be less than a fixed threshold [160]. In

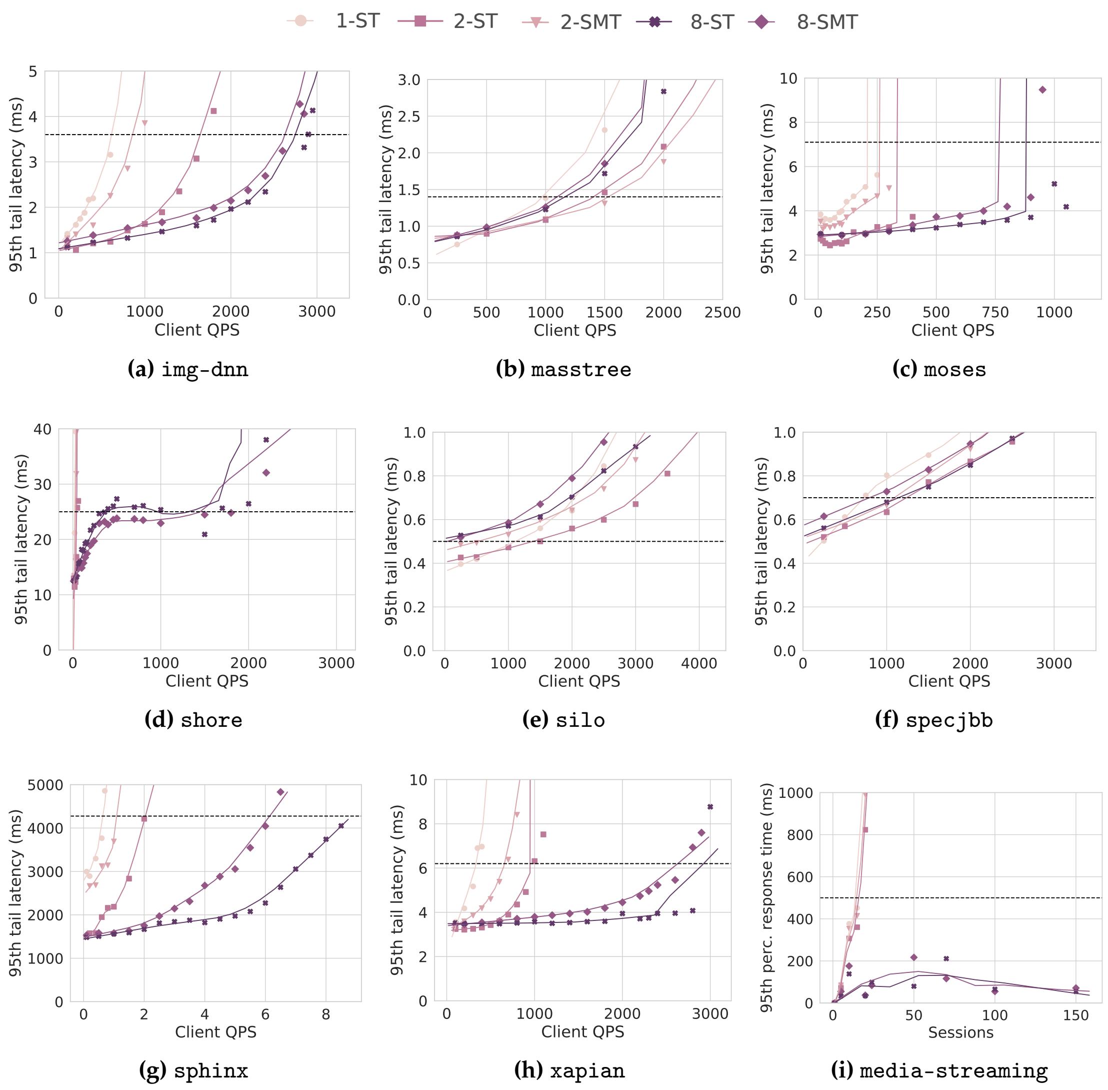


Figure 7.2: 95th tail latency (ms) for Tailbench applications and 95th percentile of the response time (ms) for `media-streaming`.

this work, we have set the threshold to 500ms as the maximum assumable delay the user should wait for the demanded streaming content.

7.2.2. Tail Latency Analysis

This section studies the effect of the studied configurations on the tail latency provided that LQoS is met. Figure 7.2[2] shows the results. The figure consists of nine plots, one for each studied application. In each plot, the LQoS value is represented by a horizontal

[2]Notice that due to QoS requirements ranging from milliseconds to seconds, different scales have been used to ease the analysis.

dotted line. Since all the studied metrics experience high variation, scatter plots are used instead of line plots. More precisely, the LOWESS [161] algorithm has been used to create a smooth trend line to ease the analysis[3]. In the analysis of this metric, we found a major observation and two main findings discussed below.

Observation 1: Performance Scalability

It is expected that the performance of the studied multithreaded latency-critical applications, which are built to execute in cloud systems, scales with the number of threads. This is true for most applications, as they exhibit great performance scalability and support higher QPS while meeting LQoS, with 2 and 8 threads, than when running on the single-threaded server. Two main tendencies can be identified in *thread scalable* applications.

Firstly, applications that significantly improve the supported QPS with both 2 and 8 threads. Applications showing this behavior are `img-dnn`, `moses`, `sphinx`, or `xapian`. For instance, `img-dnn` meets LQoS with 600 QPS, 1600 QPS, and 2900 QPS in the 1-ST, 2-ST, and 8-ST scenarios, respectively. This means that the supported QPS in the 2-ST and 8-ST is 2.67$\times$ and 4.83$\times$ higher than in the 1-ST.

Secondly, applications that show minor QPS improvements with 2 threads but exhibit a high-performance increase with 8 threads. This is the case for *shore* and *media-streaming*. For the latter application, it can be observed that the response time in both 8-ST and 8-SMT scenarios does not saturate in the same way as the other applications. The response time grows up to about 75 sessions, the point after which it remains constant and starts decreasing. Even though the LQoS point has not been reached, this trend indicates that the server has saturated.

Finding 1: QPS Insensitive Applications to the Number of Server Threads

Many factors, such as the application configuration, the application version, the system's features, and the virtualization environment, affect scalability. Therefore, adding a high number of server threads does not always translate into the server being capable of supporting more QPS. We found three applications showing this behavior. Below we analyze the reason behind this unexpected behavior for each application.

`Specjbb` supports up to 700 QPS with the single-threaded configuration and improves up to 1200 QPS with the 2-ST scenario but then experiences a little drop to 1100 QPS with the 8-ST server that quadruples the number of threads. The reason behind this behavior is, as introduced before, that the version of `specjbb` provided in the Tailbench benchmark suite has a fixed number of warehouses (a unit of stored data) during the whole run, and there is a one-to-one mapping between warehouses and threads, meaning that the number of server threads cannot be configured as in other applications. This makes the configuration of this application not optimal for scalability.

[3]We leveraged an existing implementation [162] of LOWESS, and set the alpha parameter to 0.6 and the polynomial degree to 1.

`Masstree` also supports more cumulative QPS with 2 threads than with 8 threads. We looked into the reasons that explain this behavior, and we found that the DRAM fetch cost is a limiting factor in `masstree's` scalability as also found in recent research [163]. More precisely, the number of processor stalls due to main memory accesses rises with the number of contending queries (QPS), thus the off-chip DRAM bottlenecks the performance and no improvement is obtained when increasing the number of threads. Consequently, the CPU utilization drops in the highly-threaded scenarios, as studied next.

`Silo` also shows the best results in the 2-ST scenario but is followed by the 1-ST scenario, which outperforms the 8-ST server. However, there is no consensus on the behavior of this application. For instance, in [164] `silo` is identified as scalable, and in [125], authors argue that the overhead of adding threads prevents this application from scaling beyond a few threads.

Finding 2: SMT Insensitive Applications

Since our server implements Intel processors with Hyper-Threading technology (that is, the code mark used by Intel to refer to its SMT implementation), an important decision to improve the server efficiency is whether threads should be pinned to the same physical core (i.e., to two logical cores of the same SMT core) or they should be pinned to distinct cores. If threads are pinned by couples to physical cores, only half the number of SMT cores need to be used. However, running two threads concurrently in the same core causes interference at the shared core resources [13, 165], harming the performance (i.e., QoS) with respect to single-task execution. This concern is analyzed next.

As expected, it can be observed in Figure 7.2 that for a given number of threads, the supported QPS is, in general, higher in the ST scenarios. For instance, in `sphinx`, a significant difference rises between SMT and ST configurations, which translates into 2$\times$ and 1.4$\times$ higher QPS in the 2-ST and 8-ST scenarios over the corresponding SMT scenarios, respectively. The general trend is that applications achieving higher performance in ST scenarios are those identified above, showing performance scalability with the number of threads. We use the term *SMT sensitive* to refer to these applications.

Nonetheless, it can be noticed that some applications do not benefit at all from having the threads pinned to separate cores and working in single-task mode. We refer to these applications as *SMT insensitive*. This is the case for the three insensitive applications to the number of threads, which show barely any difference between the 8-SMT and 8-ST servers since their performance does not scale. Counter-intuitively, the scalable applications *media-streaming* and *shore* also show an SMT-insensitive behavior. We looked into this event and found that the main reason behind this behavior is that the CPU is not a critical resource in these applications. In other words, reducing the pressure on the CPU resources does not translate into actual performance gains.

An interesting observation is that in *masstree* (Figure 7.2b) the 2-SMT configuration outperforms the 2-ST configuration (i.e., using only one core). We investigated this unexpected behavior and found that *masstree* keeps all the data (i.e., key-value database) in memory. These data are shared among threads. Thus, pinning two server threads to the same core reduces the core resources for each thread but improves the data sharing through the private L1 and L2 caches.

To Take Away. *In Finding 1, we have identified three workloads that do not experience scalability when running in 8-threaded scenarios and five applications showing different degrees of scalability. In Finding 2, we have identified that there is a strong connection between scalability and SMT-sensitive applications. In general, more scalable applications support higher QPS when running in single-task mode, while low scalable applications present a close SMT insensitive behavior.*

Cloud provider actions for performance improvements. *Cloud provider admins could improve resource utilization and take benefit from Finding 1 by constraining to two logical cores those server applications presenting poor scalability. The key question is how to detect these applications. Regarding Finding 2, important CPU utilization savings can be achieved in case of SMT-insensitive applications are detected at run-time.*

The major challenge for cloud providers lies in how applications can be identified as i) QPS insensitive to the number of server threads (Finding 1) and/or as ii) SMT insensitive (Finding 2) at production time. To deal with this challenge, we looked into the behavior exhibited by applications by measuring the utilization of major system components. The main purpose of this study is to analyze possible correlations between the resource consumption of the major system components and both QPS and SMT sensitivity of applications.

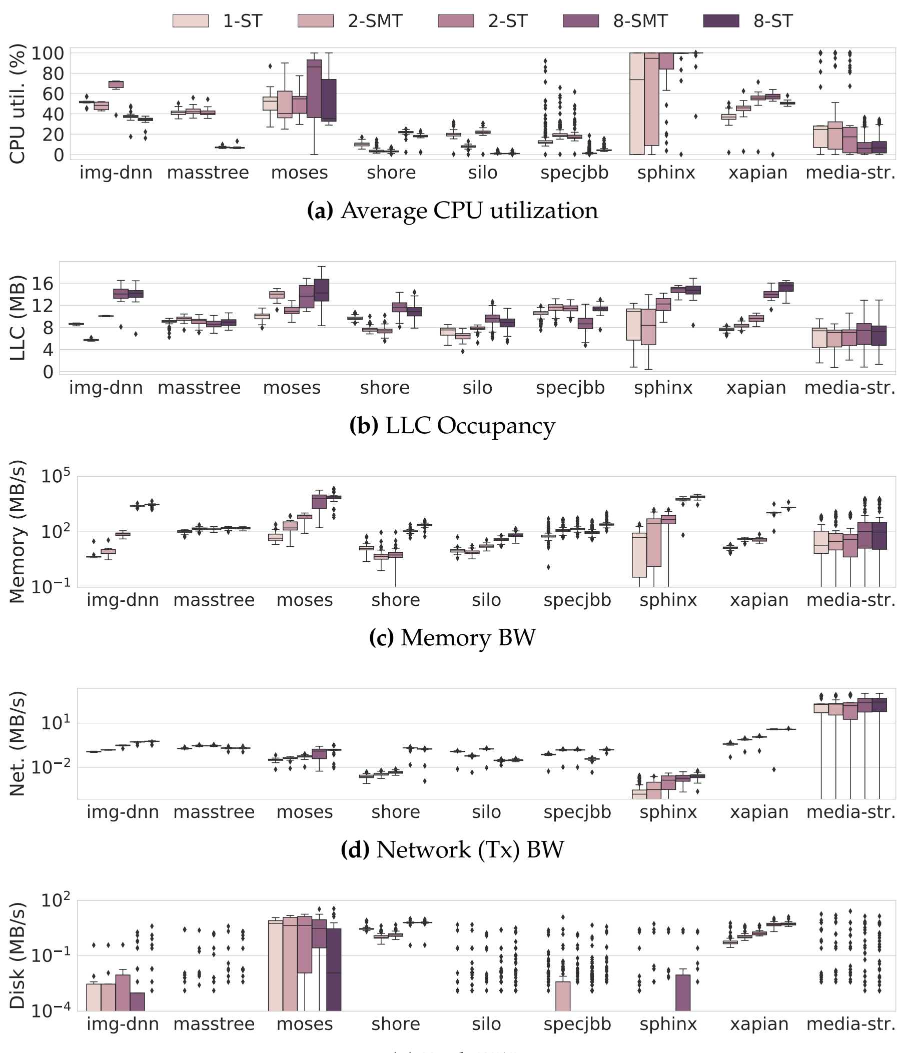
7.3 Major System Resource Consumption Analysis and Findings' Correlation

This section analyzes the utilization of the major system components (CPU utilization, LLC occupancy, main memory trend, network, and disk) for each workload and establishes logical relationships between the consumption of these resources and the findings presented in the previous section.

7.3.1. Analysis of CPU Utilization

The CPU utilization was measured at specific client load levels of interest that can be identified in Figure 7.2 as follows. It was studied at the abscissa of the point where the tail latency curve crosses the LQoS horizontal line, which represents the client load (i.e., QPS or the number of sessions in *media-streaming*) that meets the LQoS for each studied scenario.

Figure 7.3a shows the results. It plots the quartiles of the average CPU distribution with *box plots* for each scenario. Central quartiles are represented as boxes, variability outside the boxes as vertical lines, and outliers as points. A strong relationship between CPU utilization and the findings discussed above can be observed.



(a) Average CPU utilization

(b) LLC Occupancy

(c) Memory BW

(d) Network (Tx) BW

(e) Disk BW

Figure 7.3: Box plots showing the resource consumption supported by the studied applications in each scenario before reaching LQoS.

Regarding Finding 1, the three applications showing no scalability with the number of threads experience a sharp drop in CPU utilization in the highly threaded scenarios. On the contrary, this sharp drop is not experienced in applications showing scalability. Among these applications, those presenting high CPU utilization with 1-ST (e.g., `moses` and `sphinx`) further increase their utilization with 8-ST. For instance, `sphinx` increases its CPU utilization by over 90%. The reason is that high CPU utilization is caused by *memory stalls* that block the processor execution for longer, as further discussed in the next section. In contrast, those presenting medium CPU utilization (e.g., `img-dnn` and `xapian`) do not experience a CPU utilization increase in the 8-threaded scenarios, but it remains equal or even decreases.

Regarding Finding 2, SMT insensitive applications present a low CPU utilization (e.g., below 20%) regardless of the number of threads, except for `masstree` in the Tailbench applications, which presents a medium (by 40%) CPU utilization for the low-threaded scenarios. As discussed above, `masstree` presents this particular behavior because the off-chip DRAM main memory bottlenecks the system performance. Thus, adding more server threads translates into a slight decrease in the overall performance and a drop in the average CPU utilization since 4× more threads provide less system performance. In contrast, SMT-sensitive applications present a medium to high CPU utilization in the low-threaded scenarios.

7.3.2. LLC Occupancy and Main Memory Bandwidth

The shared Last Level Cache (LLC) can have a high impact on the system performance (see Chapters 4 and 5). Some recent processors have implemented hardware support that allows the system administrator to partition this component among applications at run-time (e.g., at the granularity of OS quantum). This section analyzes the space requirements of each application and its trend when increasing the number of threads. Below, we analyze the results, mainly focusing on the correlation with Observation 1.

On the one hand, applications present both different cache requirements per individual query and number of QPS; consequently, the LLC occupancy experiences wide differences among the studied applications. On the other hand, for a given application, the higher the number of queries the server processes, the higher the LLC occupation, regardless of the number of threads. However, only scalable applications can significantly raise the LLC occupancy in the 8-threaded scenarios. It is also worth mentioning that the main memory bandwidth consumed depends on the number of LLC misses. Therefore, it is strongly connected to the data locality of the cached blocks as well as the LLC occupancy since a high occupancy would rise in many LLC capacity misses.

Figures 7.3b and 7.3c present the distribution across all the execution quanta of the LLC space and main memory bandwidth consumed, respectively, by the applications in the studied scenarios at the LQoS threshold. Taking into account both LLC occupancy and main memory bandwidth, two main types of applications can be observed related to Observation 1 and Finding 1, discussed below:

1. **Applications where the 8-threaded scenarios significantly raise the supported QPS.** In these applications, the huge increase in the supported QPS translates into a huge increase in the LLC utilization, which makes these applications consume nearly all the cache capacity (over 14MB out of 16.5MB). Exceptions are *shore* (though this application consumes 12MB) and *media-streaming*, whose average LLC occupancy is around 7 MB but in some execution phases of the 8-threaded scenarios, it manages to occupy the full cache space. This common behavior is exhibited by the Tailbench applications showing scalability with eight threads. Notice that memory bandwidth increases in a 10× factor, which means that the LLC suffers a high amount of cache misses. This translates into long main memory latencies that introduce significant processor stalls and, consequently, an increase in CPU utilization.

Additionally, the relative difference in bandwidth is much higher (log scale) than the difference in LLC space, meaning that most cache misses are capacity misses.

2. **Applications showing small differences in the cache occupancy regardless of the number of threads.** This behavior can be observed in the remaining applications (i.e., those not showing QPS scalability) like *masstree*. The reason is that QPS is not improved in the 8-threaded scenarios. Consequently, the main memory bandwidth among the studied scenarios also experiences minor differences. Almost all the applications exhibiting the former behavior nearly consume all the cache space in the 8-threaded scenarios suffering a high amount of cache misses. This means that the performance of these applications is clearly limited by this processor component; hence the cloud provider should identify these applications to assign them more space. Section 7.4 analyzes this claim in further detail.

7.3.3. Disk Bandwidth

Among the studied Tailbench applications, only three of them are disk oriented: `moses`, *shore*, and `xapian`. The remaining applications make scarce usage of the disk. In fact, their disk bandwidth consumption remains below 1 MB/s during most of the execution time. Figure 7.3e presents the results.

Disk bandwidth is not constant during the execution time, but it experiences peaks and drops over time. As it can be observed in Figure 7.3e, most applications have many outlier values (represented by diamond-like dots), meaning that disk bandwidth is usually low and presents some peak values at few intervals of time. In contrast, more disk-consuming applications like *shore* include fewer outliers as the median of the disk consumption (horizontal line crossing the boxes) is much higher.

In spite of some Tailbench applications being disk oriented, the presented results make the disk bandwidth consumption not a concern in our experimental platform in terms of scalability. That is, this resource does not prevent the performance of disk-oriented applications from growing in the 8-threaded scenarios. However, remember that even if the SSD installed in the storage server allows up to 500 MB/s in large sequential reads, the bandwidth is significantly reduced when operations become small and random, and read and write operations are combined. Therefore, interference is likely to take place when multiple applications perform disk I/O operations at the same time. This issue will be studied in more detail in Section 7.4.4.

To Take Away *In this section, we have analyzed and identified the relationship between the findings and the utilization of the main system resources. We found that both non-scalable and SMT insensitive (e.g., silo) applications present medium to low CPU utilization in the low-threaded scenarios, which drastically drops in the 8-threaded scenarios. The terms low and high are used to refer to below- and above-average, respectively. On the contrary, both scalable and SMT-sensitive applications (e.g.,* `moses`*) experience a medium to high CPU utilization in the low-threaded scenarios that tend to increase in the 8-threaded scenarios. These applications present a significant increase in the main memory bandwidth consumption as the number of threads increases.*

Workload	**Resource Utilization**					**Finding 1 Affinity**	**Finding 2 Affinity**
	CPU utilization	**LLC Occupancy**	**MM trend**	**Network Bw**	**Disk Bw**	**QPS Scalability with # 8 threads**	**SMT sensitive**
`img-dnn`	Medium ≈	Medium ↑	↑↑	Low	Low	+ +	+
`masstree`	Medium ↓	Medium ≈	≈	Low	Low	No	No
`moses`	High ↑	High ↑	↑↑↑	Low	Medium	+	+
`shore`	Low ↑	Low ↑	↑	Low	Medium	+ +	No
`silo`	Low ↓	Medium ≈	≈	Low	Low	No	No
`specjbb`	Low ↓	High ≈	≈	Low	Low	No	No
`sphinx`	High ↑	Medium ↑	↑↑↑	Low	Low	+ +	+ +
`xapian`	Medium ≈	Medium ↑	↑↑	Low	Medium	+ + +	+ +
`media-streaming`	Low ↓	Low ≈	↑	High	Low	+ +	No

Table 7.3: QPS scalability, SMT sensitivity, and resource consumption of the studied applications. Resource consumption (low, medium, or high) is measured with the single-threaded server and the trend (↓ or ↑) represents the behavior change from 1-ST to 8-ST. Conditions to fulfill each finding are highlighted in yellow (Finding 1), blue (Finding 2), and green (Findings 1 and 2).

Cloud provider actions to detect applications. *In order to help cloud providers, we present Table 7.3 that shows the average utilization (low, medium, and high) of the studied resources for 1-ST, and the trend (upwards or downwards arrow) it experiences with 8-ST. The table also shows the main memory bandwidth behavior with 8-ST over 1-ST. With ↑↑↑, ↑↑, and ↑ denotes that memory bandwidth rises with 8 threads by 10^4, around 10^3, around 10^2, respectively. This table summarizes the previous findings. The conditions to fulfill Finding 1 and Finding 2 are highlighted in yellow and blue, respectively. The conditions that remain valid for both findings are colored in green. It can be concluded that only checking the CPU utilization and the memory trend is enough for cloud providers to identify the applications with higher resource requirements and carry out the corresponding actions to improve resource management previously described.*

7.3.4. Network Bandwidth

The studied TailBench workloads present negligible (i.e., below 1 MB/s) network bandwidth requirements. Thus, this consumption is not a concern in our experimental 20 Gb/s network. In contrast, `media-streaming` consumes much higher network bandwidth. Figure 7.3d presents the experimental results.

In `media-streaming`, however, bandwidth consumption is a major concern. Similarly as discussed above regarding disk bandwidth, network bandwidth experiences high peaks and drops as sessions are dynamically started and finished. This can be observed in Figure 7.3d, where the lower whisker of all *media-streaming* plots drops down to 0 MB/s and the upper whisker reaches around 1000 MB/s.

To Take Away *The network allows cloud tenants to access the cloud system. This resource is typically over-dimensioned since queuing delays in this component can force the cloud system to violate the QoS regardless of the improvement actions made in the other major system components.*

7.4 Analysis of Inter-VM Interference at the Main Shared Resources

So far, we have analyzed the performance of the server workloads by allocating threads in pairs to the same physical core or each thread to a different core. This section goes a step beyond and pursues to analyze the impact of the inter-VM (i.e., inter-workload) interference at the main shared system resources, other than the CPU. To this end, we focus on resources whose sharing can be controlled by the cloud provider. In other words, the cloud provider can apply existing advanced technologies to allocate certain amounts of specific shared resources to the running applications.

Some examples of these technologies are Intel Resource Director Technology (RDT) [166], which implements Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT) that allow monitoring and partitioning of the LLC occupancy, respectively; and Memory Bandwidth Monitoring (MBM) and Memory Bandwidth Allocation (MBA) from Intel RDT support monitoring and partitioning of the memory bandwidth, respectively. Below, we use all these technologies to limit the available space or bandwidth for the target application, and study the effect on performance [1, 54, 84]. Notice that by limiting the amount of a given resource, we mimic a scenario where the remaining fraction of the shared resource is being used by other VMs –from either the same or distinct tenant– competing for that resource.

Before analyzing the impact, we first categorize applications from the major system resource that constraints its performance.

7.4.1. Workload Classification

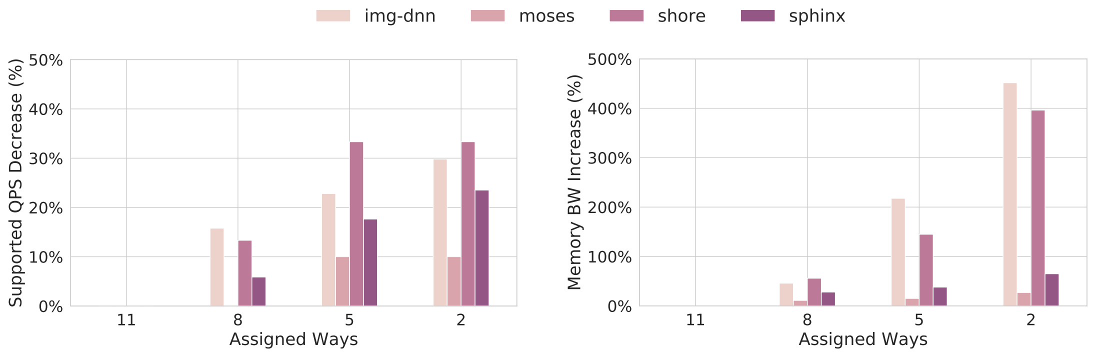
The performance of the studied workloads is mainly dominated by a major system resource (CPU, disk, or network). The studied workloads present a diversity of behaviors covering all the major system components. This section relates applications with the major consumed resources and the discussed findings.

CPU Workloads. This group includes workloads mainly dominated by CPU resources, including core, LLC, and DRAM memory, which make negligible use of network and disk. DRAM memory is included as CPU since, from the OS perspective, the time the CPU waits for DRAM accesses is accounted as CPU utilization. One could expect CPU workloads to scale in performance. However, this is not always the case since a well-balanced design and a 95^{th} tail latency large enough are required. Instead, CPU applications present both scalable and non-scalable behaviors. Examples of scalable workloads are `img-dnn` and `sphinx`, and examples of non-scalable `masstree`, `silo` and `specjbb`. The former set of applications behave as SMT sensitive and the latter as SMT insensitive.

Disk Workloads. Applications in this group present a significant disk bandwidth consumption that makes them stand out from the remaining categories. Two main categories can be distinguished according to whether the dominant bandwidth is incurred by disk read or write operations. Examples of applications presenting a medium disk utilization

CPU		Disk		Network
Non-Scalable	**Scalable**	**Read**	**Write**	**Streaming**
specjbb silo mastree	img-dnn sphinx	moses xapian	shore	media-streaming

Table 7.4: Workload Classification.



(a) Supported QPS **(b)** Memory BW

Figure 7.4: Impact of limiting the number of LLC ways on the maximum supported QPS and memory BW.

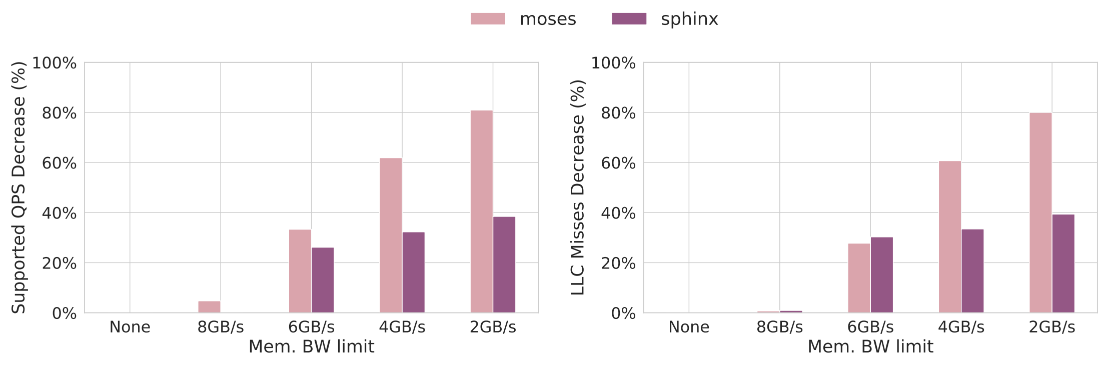
mainly due to read operations are `moses` and `xapian`, while *shore* is an example of workload presenting a medium disk utilization due to writes. An interesting observation is that all these applications scale in performance with the number of threads; however, each of them presents a different behavior with respect to SMT.

Network Streaming Workloads. As mentioned above, *media-streaming* is the only studied workload presenting a noticeable network bandwidth consumption. This application scales with the number of threads, increasing the memory bandwidth. As the number of threads increases, the CPU utilization reduces by as much as 10%.

Table 7.4 presents the devised groups (first row), categories (second row), and workload classification. Notice that all the applications except a subset of CPU workloads scale their performance as the number of threads increases. Next, we study the impact on the performance of constraining the LLC, main memory bandwidth, and disk bandwidth.

7.4.2. LLC Partitioning Analysis

This section analyzes the impact of reducing the LLC space available to the target server application. Intel CAT supports assigning specific cache ways to applications. To this end, different Classes of Service (CLOS) are defined with specific cache ways. Then, each target workload (or VM) is associated with a CLOS.



(a) Supported QPS **(b)** LLC misses

Figure 7.5: Impact of limiting the memory BW on the maximum supported QPS and LLC misses.

The Intel Xeon Silver 4116 processor, used as the server in the experimental platform, implements an 11-way 16.5 MB LLC; hence each cache way provides 1.5 MB storage capacity. To study the impact of reducing the available cache capacity to the target application, we reduced the number of cache ways assigned to the application to 7 ways (i.e., 12 MB), 5 ways (7.5 MB), and 2 ways (3 MB). The remaining cache space is assumed to be assigned to other VMs running on other cores and competing for this resource.

The study analyzes four workloads: two CPU scalable workloads (`img-dnn` and `sphinx`) and two disk workloads (`moses` and `shore`) on the 8-ST server. Figure 7.4a presents the percentage decrease of the maximum supported QPS (w.r.t. the QPS achieved with full cache space) when varying the number of LLC ways assigned to each workload. Limiting the LLC can translate into an increase in the main memory BW, especially in memory-intensive applications. This side effect can be appreciated in Figure 7.4b, which shows the memory BW increase. As observed, the applications showing the highest scalability, `img-dnn` and `shore`, present bigger QPS reduction (over 30% with 2 cache ways), and consequently, the bus bandwidth consumption significantly rises in a factor over 3.5×. Despite that `moses` and `sphinx` also present important scalability and cache occupancy, they experience lower QPS reduction than the other applications. This is due to the fact that `moses` and `sphinx` consume much more memory bandwidth under no cache constraints.

Cloud provider actions for performance improvements. *Results show that applications presenting high scalability are much more sensitive to the available cache space. Limiting this space translates into an important rise in the memory bandwidth of these applications. Therefore, the cloud provider should consider both QPS scalability and memory bandwidth as a guide to limit the cache ways assigned to a given VM.*

7.4.3. Main Memory Bandwidth Analysis

The impact of memory bandwidth on performance depends on the LLC demands of the applications and the underlying system organization. For instance, a huge LLC can catch most of the accesses of some memory-hungry applications, reducing the number of off-chip memory accesses. To carry out this experiment, we used Intel Memory Bandwidth Allocation (MBA), which works similarly to Intel CAT; applications are assigned to CLOSes since we can only limit the amount of memory bandwidth that the CLOSes can use.

For illustrative purposes, we considered the same four applications as in the previous section, and we studied the effect of reducing the memory bandwidth to 8-, 6-, 4-, and 2-GB/s on both the supported QPS and effective bandwidth consumption. The huge LLC catches most of the working set of non-memory-hungry applications when they run alone on the system, and the whole cache is available for them. Therefore, we do not analyze those applications (`img-dnn` and *shore*) having an MPKI (misses per kilo instructions) of the LLC lower than 0.4 since they are scarcely or not affected at all by constraining the memory bandwidth under these conditions.

Figure 7.5a shows the results for `moses` and `sphinx`. As observed, both applications suffer a linear degradation in the percentage of supported QPS; however, the slope of the curve in `moses` is much more pronounced, showing a much higher degradation. On average, both applications consume a similar amount of bandwidth. We looked at the reasons behind this behavior, and we found that it is mainly because `moses` has a more bursty memory bandwidth than `sphinx`, which shows a more regular pattern (see Figure 7.3c). That is, the *upper whisker* is much higher (over $10^4 \times 4$) in `moses` than in `sphinx` (around $10^4 \times 4$). Finally, limiting the main memory bandwidth slows down the execution time of the application; consequently, as a side effect, the number of LLC misses also reduces linearly with the limited bandwidth since they take place over a longer time. Figure 7.5b supports this claim. As observed, the obtained values almost match those obtained with the decrease in the supported QPS.

Cloud provider actions for performance improvements. *The memory bandwidth can have a strong impact on the performance of the workloads. The cloud provider should provide enough memory bandwidth to applications (e.g., CPU scalable) suffering a high number of LLC misses; otherwise, the performance can dramatically drop.*

7.4.4. Disk Bandwidth Analysis

The disk does not prevent the performance of disk-oriented applications from growing in the 8-threaded scenarios since the studied applications have a small consumption compared to the available total bandwidth (see Section 7.3.3). Despite this fact, the disk bandwidth is a concern in public clouds, especially in those situations where multiple VMs try to perform I/O operations at the same time.

To test this claim, we have used the microbenchmark `stress-ng` to introduce a constant stress on the disk bandwidth by performing random write operations. Different interference levels have been explored by limiting the disk bandwidth assigned to `stress-ng` to 50 MB/s, 100 MB/s and 200 MB/s, which has been done with the libvirt tool. The remaining disk bandwidth (up to 500MB/s) is available for the studied benchmark. We

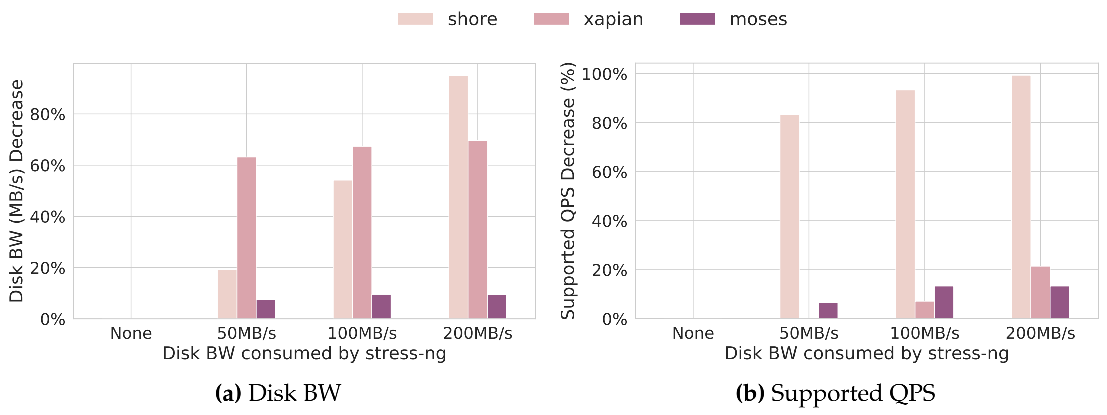


Figure 7.6: Impact of stressing the disk BW using a microbenchmark on the maximum supported QPS and reduced disk BW consumption.

analyzed the impact on the supported QPS and on how the consumed bandwidth reduces over isolated execution.

Figure 7.6 shows the results for the three disk applications, `shore`, `xapian`, and `moses`. Figure 7.6a shows the impact of `stress-ng` on the consumed bandwidth of the disk applications for the studied interference levels. The consumed bandwidth significantly reduces over isolated execution (*None*), especially in `shore` and `xapian`. This happens because their bandwidth consumption keeps similar across all the execution intervals; that is, the standard deviation of the gathered bandwidth values is very small, and all the values fall close to the median (see Figure 7.3e), and `stress-ng` reduces the available bandwidth across all the execution time. In contrast, `moses` consumes most of its bandwidth at the beginning of its execution; thus, its consumption is only affected at that execution phase.

Figure 7.6b shows the impact of the bandwidth reduction on the supported QPS. It is strongly related to the type of disk operation performed. It can be noticed that disk read workloads (i.e., at least 80% of the operations are reads) suffer less performance degradation, even when their bandwidth decreases over 60% in `xapian`. In this case, performance drops by 20% when `stress-ng` consumes 200MB/s. In contrast, in disk write workloads like `shore`, the supported QPS decrease (in percentage) is higher than the bandwidth consumption decrease.

Cloud provider actions for performance improvements. *There is a strong connection between disk bandwidth consumption and the performance of disk applications, especially for write disk workloads that present a homogeneous disk consumption across the execution time. The cloud provider should take special care with these applications by pinning them to machines with balanced or low disk bandwidth consumption.*

7.5 Summary

This chapter has presented a characterization study of cloud workloads aimed at identifying key relationships between performance and resource consumption. The results show that CPU resources can be significantly reduced by considering that the performance of some applications does not scale with the number of threads and that threads of Hyper-Threading insensitive applications can be allocated to the same physical cores without affecting performance. Identifying these applications at run-time, however, is challenging. We have shown that this challenge can be successfully dealt with by analyzing the utilization of the major system components. In addition to CPU, we have also studied how each application's share of other major shared system resources impacts performance. Experimental results show that some applications can suffer performance losses of over 80% if not provided with a big-enough share of its critical resource.

The conclusions of the presented studies and the discussions identifying cause-effect relationships among the utilization of the different system components can be used as a basis for cloud providers to develop virtualization strategies.

CHAPTER 8

Detecting and Estimating Inter-VM Interference in the Public Cloud

Cloud providers can achieve economies of scale by building large-scale data centers and sharing resources among different jobs which run in virtual machines (VMs). However, sharing the physical machine means that inter-VM performance interference will appear due to multiple VMs competing for the major system resources (e.g., processor cores, last level cache (LLC), or main memory), making performance unpredictable and jeopardizing compliance of the service level agreement (SLA) between a cloud provider and a customer.

To avoid QoS violations and tackle inter-VM performance interference, cloud providers typically adopt an *overprovisioning* strategy, where resources are assigned to each VM in excess to avoid possible performance degradation due to inter-VM interference. Moreover, public cloud servers frequently run latency-critical workloads. Unlike conventional workloads, the performance of latency-critical applications is given by the obtained tail latency, indicated as a percentile (e.g., 95^{th} or 99^{th}) of all the latencies and accounts for the requests that take longer to complete. This means that the performance of these workloads is very sensitive to inter-VM interference. Because of this fact, system resources need to be conservatively overprovisioned to ensure compliance with the SLA. This workaround, however, results in poor utilization of the major resources of the cloud system.

This chapter proposes Cloud White, an approach that can detect and quantify inter-VM interference online when running multiple latency-critical applications in different VMs running on the same physical machine. With our approach, the VM *is not* a black box anymore, but its behavior (and introduced interference) is revealed, becoming “white” or Cloud White.

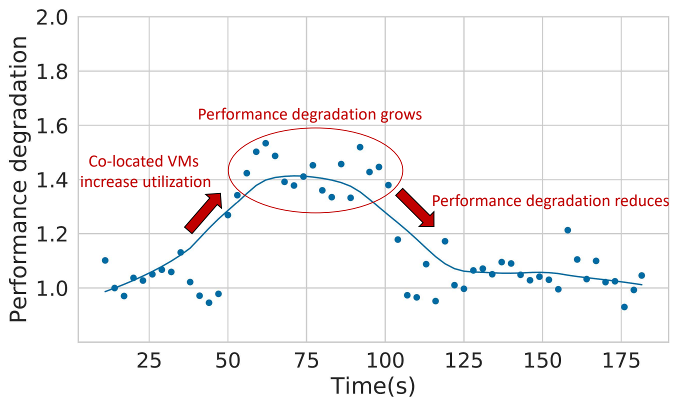


Figure 8.1: Typical scenario: 1) co-located VM load grows and increases interference, 2) performance degradation increases for the target VM, 3) co-located VM reduces its load.

8.1 Motivation

In a public cloud system, a VM can experience performance losses due to a change either in its load (*intra-VM*) or in the load of other VMs (*inter-VM*). Detecting the cause of interference inducing the performance loss is challenging. The former case refers to intra-VM interference, and thus, it is not a concern for the cloud provider, which is the focus of this manuscript. This chapter concentrates on the latter case, where two or more VMs are involved, and the performance degrades due to the inter-VM interference. We use the terms *victim* and *inflicting* to refer to VM(s) suffering performance degradation and the VM(s) increasing its load, respectively. Notice that a given VM can become a victim or act as inflicting across different periods of its execution time. The main goal of this chapter is twofold: detect the victim VM(s) and estimate its performance degradation, both especially challenging when all the co-running applications are latency-critical in the public cloud.

We illustrate the problem in Figure 8.1, which shows how the tail latency of a *victim* VM co-located with multiple resource-consuming VMs degrades over a 180-second time interval. The example starts in a steady situation with all co-located VMs running at a relatively low utilization (i.e., shared resources are not stressed). This steady situation is assumed to be the *normal conditions* or baseline situation; thus, we assume that at this point in time, the performance of the victim VM is not (or slightly) suffering due to the inter-VM interference (1 means no performance degradation). At second 50, one of the co-located VMs (*inflicting* VM) increases its load and starts consuming more shared resources. This load increase impacts the performance of other VMs. In the example, the victim degrades up to $1.4\times - 1.6\times$ its performance compared to the steady situation. After the second 100, the resource consumption of the inflicting VM reduces, and the performance degradation of the victim VM returns to a low level.

Discerning which are the victim and inflicting VMs is challenging in the public cloud, especially when running multiple latency-critical applications, since i) VMs are seen as

Approach	Year	Main Features				
		Interference detection	Performance interference prediction	Tail latency	All main system resources?	Black box?
Deepdive [91]	2013	✓, low-level metrics	✓, but in isolation	✗, average	✓	✓
RC [36]	2017	✗	✓, VM information	✓, median / 99th	✗, no disk & network	✓
Scavenger [32]	2018	✓, VM resource usage	✗	✓, 90th / 95th	✗, no disk	✓
Alita [29]	2020	✓, low-level metrics	✗	✗, average	✗, no disk & network	✓
Twig [99]	2020	✗	✓, low-level metrics	✓, 99th	✗, no disk & network	✗
CLITE [30]	2020	✗	✓, resource shares	✓, 95th	✓	✗
Cloud White	2022	✓, GIPS metric	✓, low-level metrics	✓, 95th	✓	✓

Table 8.1: Summary of closely-related work to Cloud White supporting latency-critical applications, sorted by chronological order.

"black boxes" so the cloud provider has no information about the applications running on the VMs, and ii) these applications present rather low shared resources utilization.

Some previous works have tried to address these challenges. Table 8.1 summarizes the main features of these approaches. However, these approaches present two main shortcomings. On the one hand, performance interference is detected in a subset of the system resources. On the other hand, no prediction is made on how performance interference impacts on the QoS of VMs.

To overcome these issues, this chapter proposes Cloud White, an approach that not only identifies the victim VM(s) in cloud systems running multiple latency-critical applications as "black boxes", but also provides accurate estimates of their performance degradation. Cloud White uses the Giga Instructions Per Second (GIPS) to identify the victim VM(s). The novelty does not lie in the GIPS metric itself, but we show its effectiveness for this purpose; in other words, as far as we know, it is the first time it is used to discern the victim VM(s). Performance degradation is estimated through multi-variable regression models.

Cloud White is, to the best of our knowledge, the first approach that can deal with the two challenges mentioned above: identify the victim VM(s) in cloud systems with latency-critical applications, and provide accurate estimates of its performance degradation. Moreover, the dynamic prediction error is, on average, below 10%.

8.2 Experimental Setup

8.2.1. Platform

With the aim of building a controlled environment closely resembling one of the public clouds, we set up an experimental platform described in detail in Chapter 3.2.

In summary, our experimental platform includes all the components of a typical cloud system [167]. Even though it is not composed of many nodes like a production system, the setup of one server node and one client node complies with the twofold objective of this work: i) study the interference among VMs executing in the same server node, and ii) use the client node to carry out experiments with a controlled load that helps us evaluate (even saturate the server) and validate our approach.

8.2.2. Workloads

Latency-Critical Benchmarks

In this work, we use a set of representative latency-critical benchmarks from the Tailbench suite [125]: `img-dnn`, `masstree`, `moses`, `silo` and `specjbb`. More details can be found in Chapter 3.4. According to the characteristics of these applications, our results confirm that they are mainly limited by CPU or memory resources (more details in Chapter 7); thus, two different models will be required to predict the performance (see Section 8.4).

Microbenchmarks

To introduce interference in a controlled manner, a practical option commonly employed [50, 32, 31, 29, 94] is the use of microbenchmarks, or synthetic benchmarks, specially designed to stress specific systems resources (e.g., the cache, the main memory, or the disk) by introducing interference in their utilization. One of the main advantages of using microbenchmarks is that they can be finely tuned, setting different utilization levels on each shared resource. To generate a workload that introduces realistic interference, we have used the *stress-ng* [168] microbenchmark, which stresses the three main system components (LLC, main memory bandwidth, and disk). In addition, we used *iPerf3* [169] to stress the network bandwidth.

Below, we present five types of workload scenarios created with the microbenchmarks to analyze different stressing levels at the major system components.

No interference. This scenario represents the common situation of the target system, where there is a low (by 10%) processor utilization and resource usage. It will be used as our baseline scenario to check performance degradation.

Cache-memory stressing. This scenario is used to study the effect of stressing the LLC occupancy and main memory bandwidth. Three main stressing levels have been considered: memory bandwidth consumption from 1.9 to 5.5 GB/s and LLC space from approx. 12.5 to 14.7 MB.

Disk stressing. This scenario implies a high number of write and read operations to disk. Three stressing levels have been considered, from moderate (around 45 MB/s) to high (over 100 MB/s) disk bandwidth consumption.

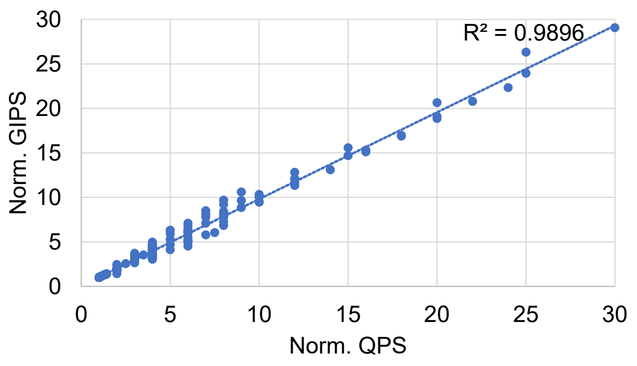


Figure 8.2: Relationship between GIPS and QPS for the five studied latency-critical applications.

Cache-memory-disk stressing. This scenario adds the stress introduced by the cache-memory and disk scenarios within the same experiment.

Network stressing. This scenario stresses the network bandwidth (both receive and transmit, but in separate experiments).

8.2.3. System Load Conditions

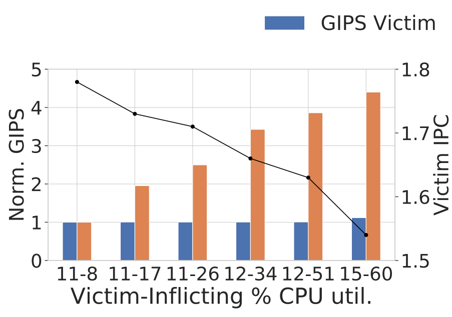
A key modeling decision is to establish the load level of interest to be studied. The client load is controlled by varying the queries per second (QPS) performed by clients. However, the QPS is an internal metric of the workload unknown to the cloud provider. Thus, we consider CPU utilization as a proxy of the load from the cloud provider perspective.

To this end, two main CPU utilization levels are of interest to the cloud provider: normal and overloaded conditions. Normal conditions refer to low processor utilization (e.g., 10%), where the interference introduced due to the VM load is negligible. Current cloud servers [36, 37] typically run at this low CPU utilization. Overloaded conditions refer to a relatively high processor utilization (e.g., over 50%), where cloud providers assume the system SLO may be violated. It is important to note that the utilization value strongly depends on the type of workload. For instance, if performance is measured in terms of tail latency, minor CPU variations can turn into a significant tail latency increase and cause important performance degradation (more details in Chapter 7). We considered CPU utilizations ranging from 10% to 60% to analyze the impact of varying the inflicting VM's load on the victim's performance.

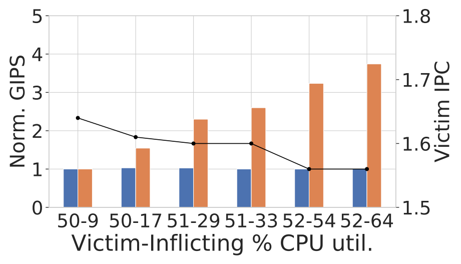
8.3 Cloud White: Detecting the Inter-VM Interference

8.3.1. GIPS: a New Way to Detect the Victim and Inflicting VMs

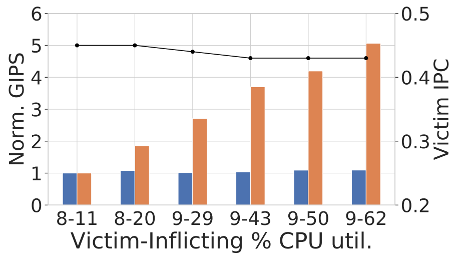
One could think that CPU utilization could be used as an intuitive solution to detect the VM causing the interference. The CPU utilization, however, grows not only when



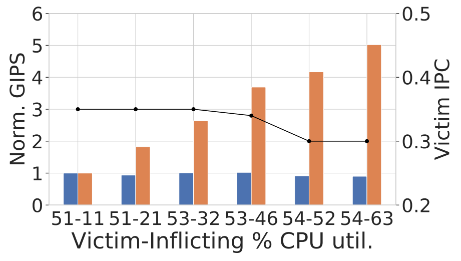
(a) Victim VM `img-dnn` (Ucpu low)
Inflicting `specjbb`



(b) Victim VM `img-dnn` (Ucpu high)
Inflicting `specjbb`



(c) Victim VM *specjbb* (Ucpu low)
Inflicting `img-dnn`



(d) Victim VM `specjbb` (Ucpu high)
Inflicting `img-dnn`

Figure 8.3: Normalized GIPS of victim VM (fixed QPS) and inflicting VM (increasing QPS) for different CPU loads, high and low. The right Y-axis shows the IPC obtained (black line) by the victim VM in each of the test cases.

the QPS of the target VM grows during its execution but also due to the inter-VM interference. The reason is that the CPU utilization comprises both the time the processor executes instructions, which grows with the QPS, and the time it is stalled waiting for memory accesses, which grows with interference. This means that, apart from CPU utilization, other metrics are needed, as discussed below.

To cope with this challenge, a wide set of performance metrics (like the IPC, ROB stalls, etc.) were analyzed. We found that the number of instructions executed per second helps identify the inflicting and victim VMs. More precisely, the **Giga Instructions executed Per Second** (GIPS). Based on this finding, the devised approach uses the GIPS to identify the victim VM(s). Notice that the novelty is not the metric itself but the purpose for which it is used. The rationale is as follows. The amount of instructions a VM commits per second is directly related to the number of queries it receives per second (QPS). That is, if a victim VM keeps its load steady (i.e., the server processes the same requests per second), its GIPS remains unaffected. Otherwise, the GIPS will change.

To support this claim, we studied the correlation between QPS and GIPS. Figure 8.2 plots the results, varying the QPS for the five studied applications executed under four server threads (1, 2, 4, and 8) configurations. Tested QPS ranges from 100 to 2000 (i.e., the maximum value before the 95th tail latency saturates). For each server configuration, both QPS and GIPS have been normalized to that of the lowest QPS value (i.e., 100). A total of 175 points are plotted. Results show that the GIPS presents a very strong linear correlation (R^2 = 0.9896), almost perfect, to QPS.

It should be taken into account that cloud workloads do not stress 100% the CPU, but the CPU utilization usually takes low values (e.g., 20%) [36, 37]. Therefore, IPC and GIPS are not directly correlated since IPC is calculated over the time the CPU is busy (i.e., utilization time) while the GIPS is computed over the total measurement time, which comprises both the utilization time and time the VM is waiting for client requests.

8.3.2. Case Study with Two VMs

This section shows how the GIPS helps detect the victim and the inflicting VMs running latency-critical applications through a practical example. For illustrative purposes, only two VMs are used, one acting as the victim and another as the inflicting.

Figure 8.3 shows the GIPS and the IPC varying the CPU utilization of the victim and inflicting VMs across four different scenarios. In each plot, each pair of bars corresponds to a different experiment. The X-axis legend below each bar indicates the CPU utilization of the victim and inflicting VMs in the experiment. Two load (utilization) levels, low (left-side plots) and high (right-side plots), are considered for the victim VM, which refer to about 10% and 50% CPU utilization in isolation, respectively. The utilization of the inflicting VM has been set to grow from about 10% (no contention) up to 60%. More precisely, QPS has been experimentally configured to achieve that utilization. The first pair of columns refer to a low contention scenario where the inflicting VM presents a low load, and it is used as a baseline. The GIPS of both VMs are normalized to the achieved in the baseline scenario.

In the upper plots, `img-dnn` is the victim VM under low load (Figure 8.3a) and high load (Figure 8.3b), and `specjbb` is the inflicting VM growing the interference from left to right. This can be appreciated as the normalized GIPS of the inflicting VM (orange bar) grows by 4$\times$ in the highest interference. On the contrary, the GIPS of the victim VM (blue bar) remains around 1 across the six experiments in the plot. The interference introduces an important change in the victim's behavior. The CPU utilization grows from 11% to 15% when it works under low load (left side plot) and from 50% to 52% (right side plot). This represents an important growth (in percentage), especially under low load conditions, thus, it translates into significant performance losses. Notice that the IPC of the victim VM (right Y-axis) significantly decreases as the interference increases, dropping from around 1.78 to 1.53.

In the bottom plots (Figures 8.3c and 8.3d) the applications change their role: `img-dnn` is the inflicting VM and `specjbb` is victim VM. Similar results can be observed. Again, the GIPS of the victim VM are not affected as the interference increases while the GIPS of the inflicting VM grows up in a 5$\times$ factor. Unlike the previous figures, the CPU utilization of the victim grows in a negligible way (less than 1%); consequently, its IPC drops to a lesser extent, especially in the low load plot (Figure 8.3c). This is because `img-dnn` is more memory-intensive than `specjbb` and, thus, is more sensitive to the co-runner's workload.

Algorithm 3 Cloud White's interference detection phase pseudo-code where inflicting and victim behaviors are detected.

```
1: for all $apps$ do
2:     Compute $\%guest_{norm}$ and $GIPS_{norm}$
3:     if $\%guest_{norm}$ increases M% then
4:         if $GIPS_{norm}$ increases N% then
5:             Inflicting behavior
6:             Tag as victim all non-inflicting VMs
7:         else
8:             Victim behavior
9:         end if
10:    else
11:        VM's behavior is steady
12:    end if
13: end for
```

Finally, we would like to remark that the GIPS metric is not limited to two VMs, but it can be applied regardless of the number of VMs (see Section 8.5.3).

8.3.3. Cloud White's Interference Detection Phase Algorithm

Algorithm 3 presents the pseudo-code of the interference detection mechanism performed by Cloud White. That is, the steps carried out to discern if a given VM presents an inflicting or a victim behavior. First, the normalized load and GIPS are computed for each application. Load is quantified in terms of percentage of *guest time* (from the `/proc/stat` system file, see Section 8.4.2). Normalized values are computed with respect to *No Interference* scenario (see Section 8.5.1 for more details). Once computed, Cloud White checks if the VM has experienced a load change. This is done by checking if *%guest time* has changed –increase or decrease– more than a given threshold ($M\%$) over the rolling mean of the previous intervals. If this condition fulfills, then it checks if the GIPS has also noticeably changed (over $N\%$). In such a case, the VM is tagged as inflicting. Otherwise, it is tagged as a victim. Notice that some VMs may not experience load changes (lines 10-11). However, if one of the co-located VM(s) exhibits an inflicting behavior, the VMs exhibiting a steady behavior are also tagged as victims as they may be possibly affected in the near future by the interference caused by the inflicting VM(s). The results presented in this work have been obtained with $M\%$ and $N\%$ set to 5 and 50, respectively.

8.4 Cloud White: Modeling Performance Degradation

8.4.1. Experimental Methodology

To devise the models to estimate the performance degradation, we first need to study the impact of the interference on the system performance. This study needs to be done in a controlled way; that is, we need a method to control the introduced interference. To this end, we have used microbenchmarks described in Section 8.2.2 to create typical scenarios of interference in the system's main shared resources.

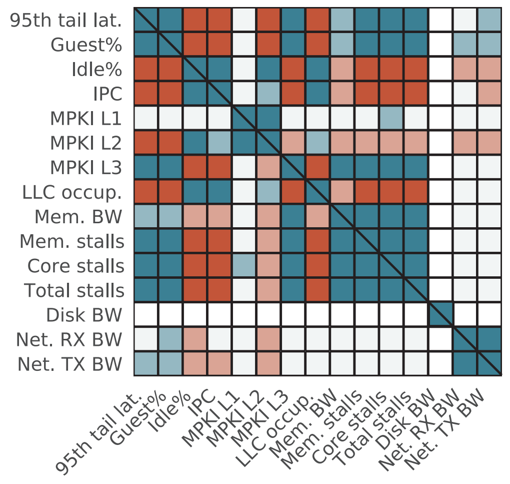
In the experiments, all (24 logical) cores of our machine are used to model scenarios closely resembling to a production system. Two different instances of the microbenchmark are used. One instance is devoted to stress one or some shared resources and is hosted in a VM with 8 logical cores. The other instance referred to as *no interference*, has been used to introduce relatively low interference that will barely affect the victim VM. To this end, 6 logical cores are occupied, presenting each of them with a low utilization. In this way, the stress introduced comes only from a subset (i.e., 8) of the system's cores, as it is likely to occur in real systems. From the remaining logical cores, 2 were devoted to the victim VM and 8 to run the software agent components (OVS, DPDK, and Stratus manager described in Chapter 3.2). From now on, we assume the same core distribution of the software agents.

The devised models should be general enough to adapt to other applications showing similar characteristics. To check this claim, we make two different groups of workloads: one for building the models (*known workloads*) and another for evaluating the models (*unknown workloads*) not previously used. From our set of applications, we identified two behaviors, CPU- and memory-bound (see Section 8.2.2). In case Cloud White encounters an application presenting an unseen behavior, models would be trained with this application to update or generate new models. We use `specjbb` and `img-dnn` as known workloads presenting CPU- and memory-bound behaviors, respectively, and leave `silo`, `masstree` and `moses` as unknown applications.

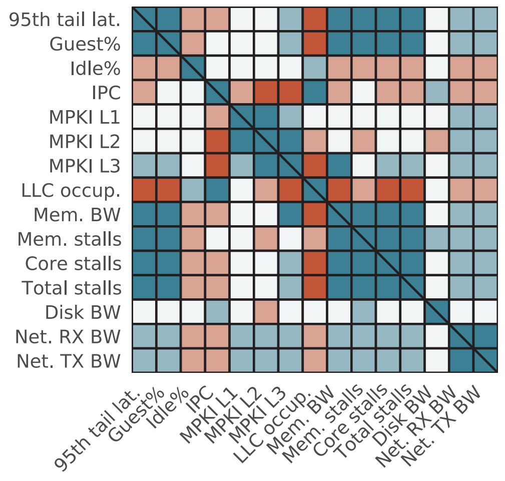
8.4.2. Looking for Performance Metrics as Model Parameters

Since the cloud provider sees tenant VMs as "black boxes", we studied a wide set of performance indicators from the main system components (processor, memory, network, and disk) with the aim of finding the potential correlation between them and performance degradation (i.e., tail latency).

We evaluated two different CPU metrics related to where the processor spends time: *guest* time, which accounts for the time spent by processes running on a virtual CPU, and *idle* time, which represents the time spent idling (i.e., not executing instructions) while there are no disk I/O requests outstanding. To assess the core performance, we have evaluated metrics to quantify throughput (e.g., IPC) and interference within the core (e.g., processor stalls and L1 cache misses). In addition, we have studied metrics that quantify the LLC occupancy and bandwidth consumption in the main memory, disk, and network. Regarding network bandwidth, we found out that the Tailbench applications consumption is very low compared to the maximum supported bandwidth (over 20 Gb/s) in our experimental platform. Thus, no performance degradation was observed due to the interference in the network bandwidth. Furthermore, the cloud provider should always provide enough network bandwidth to almost completely avoid interference at this component because queuing delays at the network will immediately prevent the application from meeting the required QoS.



(a) `img-dnn`



(b) `specjbb`

Figure 8.4: Correlation of the studied metrics for `img-dnn` and `specjbb` for a 20% processor utilization.

Figure 8.4 illustrates the correlation between the degradation of the 95^{th} tail latency and the studied metrics (left-most column and top row) for the two applications representative of memory- and CPU-bound behaviors, `img-dnn` and `specjbb` running a QPS that results in an average CPU utilization of 20%. That is, how much performance degradation (compared to that obtained in the *No Interference* scenario) is related to each of the studied performance metrics. Additionally, the correlation between each pair of metrics is shown. Each table represents the correlation of the studied metrics for each application, obtained from the results of all the experiments performed with the microbenchmarks in all the stressing scenarios previously discussed. Positive correlations are colored in blue and negative ones in red, where a darker shade implies a stronger correlation. Correlations between 0.2 and 0.6 are colored in light blue, and correlations between 0.6 and 1 are in dark blue. Negative correlations within the same range are colored in light and dark red. Cells colored in white represent a very low correlation (i.e., between -0.2 and 0.2). As observed in the *95th tail lat.* column, `img-dnn's` performance strongly correlates with the processor utilization, core, and LLC metrics. However, `specjbb` achieves the strongest correlation in just a subset of these metrics (percentage of guest time and processor stalls) and the main memory bandwidth. Even though `specjbb` is CPU-bound, memory bandwidth has a stronger correlation with performance degradation in `specjbb` than in `img-dnn`. `Specjbb`'s tail latency is 4× shorter than that of `img-dnn` and consumes more memory resources, so a small impact on the memory bandwidth can turn into performance degradation. Note, however, that the correlation factor indicates how related are the variations in both metrics but gives no insight into the impact on the performance of the interference in that component. In both cases, network- and disk-related metrics achieve a poor correlation, being stronger in `specjbb` but not significant enough. This was expected because the two workloads present relatively low disk and network utilization.

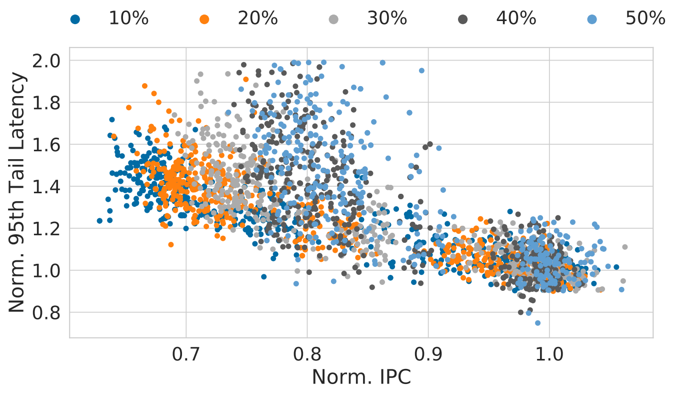


Figure 8.5: Relationship between IPC and performance degradation for `img-dnn` varying the CPU utilization (10% to 50%).

8.4.3. Relationship between System Load and the Models

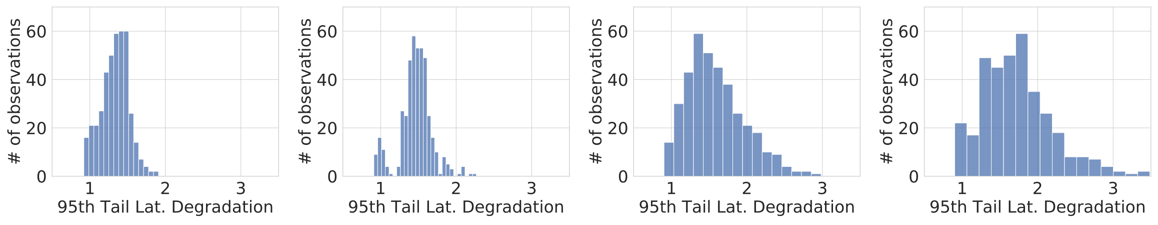
So far, we have considered that the CPU utilization of the victim remains constant and the CPU utilization of the inflicting ranges from 10% to 60% (higher utilization normally translates into higher interference) to analyze workload conditions of interest (see Section 8.2.3). However, in the real cloud, the processor utilization of the victim does not remain constant, but it tends to vary in the range from normal to overloaded.

Under different CPU utilization levels, we observed that the trend of the performance metrics also varies. An example can be observed in Figure 8.5, showing the relationship between dynamic performance degradation (95th tail latency) and IPC for the studied CPU utilization scenarios for `img-dnn`. Each point corresponds to the 95th tail latency and IPC achieved over a time interval of 5 seconds, resulting in more than 1000 points plotted in the graph.

Values have been normalized to the average value obtained with the *No Interference* workload scenario (see Section 8.2.2). As observed, the slope of the points corresponding to each CPU utilization (with different colors) varies, being more pronounced for higher processor utilization. Additionally, points that belong to higher CPU loads (i.e., 40% and 50%) present a non-linear trend. Therefore, building a unique model that embraces all processor utilizations may lead to high prediction errors. To deal with this fact, specific models can be built for distinct processor utilization (e.g., 10%, 30%, and 50%).

8.4.4. Multi-Variable Regression Models

To acquire sound knowledge about the performance-related metrics (see Section 8.4.2), we analyzed how each metric individually correlates with performance degradation using regression models. We found that none of them strongly correlate with performance in a generalized way. Therefore, we looked into multi-variable regression models to devise models that achieve a stronger correlation. These models pursue improving the correlation by combining several metrics.



(a) `Img-dnn` - CPU 20% **(b)** `Specjbb` - CPU 20% **(c)** `Img-dnn` - CPU 50% **(d)** `Specjbb` - CPU 50%

Figure 8.6: Histograms with the frequency distribution of the 95th tail latency degradation values in different scenarios.

Designing statistical models. Due to the non-linear nature of the tail latency metric, linear regression models are not the most suitable models to use. We also observed that the performance degradation metric does not follow a normal distribution.

Figure 8.6 shows the distribution of the performance degradation (95th tail latency normalized over *No interference*). A value of 1 on the X-axis means no performance degradation. Results were gathered in the stressing scenarios for `img-dnn` and `specjbb` for two processor utilization levels: 20% (upper plots) and 50% (bottom plots). As observed, the histograms do not exhibit a symmetric shape. As expected, in the top plots where the CPU utilization (20%) is lower, the performance is less affected by the interference. This can be appreciated in that, in around 96% of the observations, performance degrades by a factor of 1.9×. In contrast, in the highest CPU load scenario (Figures 8.6c and 8.6d), this range is extended up to around 2.8, meaning that in some cases, performance degradation can be as much as 180%.

In all cases, the histogram shape does not resemble a typical bell-shaped curve but is right skewed, especially in the highest CPU load plots, illustrating that the Gaussian distribution is not the most appropriate distribution for tail latency [170]. To check this, we modeled the measured data linearly and performed the Anderson-Darling test [171], which confirmed that, generally, the models' residuals were not normally distributed. We found that only `img-dnn's` 50% CPU load model satisfied the normality assumption, so this can be considered as a rare case. Additionally, we checked that in most cases, performance degradation did not present a linear relationship with the studied metric (e.g., IPC values corresponding to 40% and 50% CPU utilization in Figure 8.5). Therefore, for generalization purposes, we looked into other models that do not require the data to be normally distributed. Among these, we found that the generalized linear models (GLM) [172] are the most appropriate for non-linear and skewed distributions [173].

Model fitting - reduction strategy. The models were built and fitted using *statsmodel* Python library [171]. The model fitting was performed with the iteratively reweighted least squares (IRLS) method, with the objective of minimizing the deviation that occurs when estimating performance degradation. No constraints were specified to obtain the model coefficient values. A model reduction strategy [174] by statistical significance of the terms has been followed to find the truly significant variables. This implies that,

Generalized Linear Model Regression Results			
Dep. Variable:	perf. deg.	**No. Observations:**	425
Model:	GLM	**Df Residuals:**	420
Model Family:	Gamma	**Df Model:**	4
Link Function:	inverse_power	**Scale:**	0.0043907
Method:	IRLS	**Log-Likelihood:**	396.96
No. Iterations:	6	**Deviance:**	1.7979
Covariance Type:	nonrobust	**Pearson chi2:**	1.84

	coef	**std err**	**z**	$P > \|z\|$	**[0.025**	**0.975]**
const	1.0602	0.017	63.261	0.000	1.027	1.093
guest%	-0.0207	0.009	-2.261	0.024	-0.039	-0.003
Mem. BW	-0.0233	0.002	-10.432	0.000	-0.028	-0.019
Core stalls	-0.1674	0.017	-9.713	0.000	-0.201	-0.134
Total stalls	0.0876	0.016	5.446	0.000	0.056	0.119

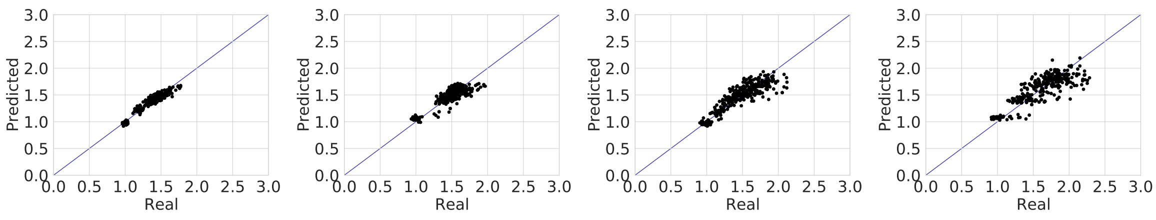
Table 8.2: Example of the parameters and statistics of the prediction model generated with data from `specjbb` with a 20% CPU load.

initially, the model is built including all candidate variables (performance metrics), and the least statistically significant variable (largest `P>|z|`, fulfilling that is higher than 5%) is removed. This process is repeated until the model only contains significant terms. In this work, we considered candidate variables, the performance metrics achieving a correlation higher than 0.4.

For illustrative purposes, Table 8.2 shows an example of the model generated with the data from the experiments performed with `specjbb` under stressing conditions for 20% CPU utilization. The upper part of the table shows a summary of the characteristics of the resultant model, both qualitative and quantitative. In this example, the obtained model belongs to the GLM Gamma exponential family, with inverse_power (i.e., reciprocal) link function, the default link function for the Gamma family. Other link functions were explored, like the logarithmic function, but the best results were obtained with the inverse_power. We would like to remark that the model characteristics are the same for the models generated for all the CPU utilization levels. Results like the `Log-Likelihood`, `Deviance`, and `Pearson chi2` [175] indicate the model goodness to the data used to train it. For instance, deviance is a measure of goodness of fit (the lower and closer to zero, the better).

The bottom part of Table 8.2 shows the coefficients of the model variables, as well as the standard error of each one. Among the generated models, this is the part that mainly differs since each model has different (both in number and value) coefficients. As a result of the model reduction, all independent variables present a `P>|z|` lower than 5%. In this example, the model comprises five independent terms: a constant term and four significant variables (degrees of freedom, Df, equals 4), which correspond to the metrics guest%, memory BW, core stalls, and total stalls.

The goodness of the model can also be analyzed graphically. Figure 8.7 shows four examples of the relationship between the real performance degradation values with the predicted values for `img-dnn` and `specjbb` in two different CPU utilization scenarios. Results



(a) `img-dnn` - CPU 20% **(b)** `specjbb` - CPU 20% **(c)** `img-dnn` - CPU 50% **(d)** `specjbb` - CPU 50%

Figure 8.7: Scatter plots with real (measured) vs. predicted performance degradation.

show that most of the points lie on the curve $x = y$ (plotted diagonal) or very close to it, which indicates the *perfect prediction* (i.e., real and predicted values are equal). Higher processor utilization scenarios (Figure 8.7c and Figure 8.7d) show the higher spread, something expected since tail latency is more affected with higher CPU utilization (i.e., overloaded scenario).

Final models. This section presents the memory-bound and CPU-bound models devised in this work for `img-dnn` and `specjbb`, respectively. Equations 8.1 and 8.2 show the formulas where K, $x1$, $x2$ to xn represent the coefficients of the constant term and the independent variables, respectively. y_{mem} and y_{CPU} represent the dependent variables in terms of performance degradation for the memory-bound and CPU-bound models, respectively.

$$y_{mem} = \frac{1}{\begin{pmatrix} K + (x1 \times idle\%) + (x2 \times IPC) \\ +(x3 \times MPKI\_L2) + (x4 \times MPKI\_L3) \\ +(x5 \times mem\_BW) + (x6 \times mem\_stalls) \\ +(x7 \times core\_stalls) + (x8 \times total\_stalls) \end{pmatrix}} \tag{8.1}$$

$$y_{CPU} = \frac{1}{\begin{pmatrix} K + (x1 \times guest\%) + (x2 \times idle\%) + (x3 \times IPC) \\ +(x4 \times mem\_BW) + (x5 \times MPKI\_L2) \\ +(x6 \times MPKI\_L3) + (x7 \times mem\_stalls) \\ +(x8 \times core\_stalls) + (x9 \times total\_stalls) \end{pmatrix}} \tag{8.2}$$

Since performance degradation strongly depends on the CPU utilization, each model needs to be tuned for five CPU values (i.e., 10%, 20%, 30%, 40%, and 50%). We found that this number is enough to provide accurate estimates. For each individual model (i.e., the model generated for a CPU load), only those statistically significant terms have been considered, meaning the coefficients of the remaining terms are set to zero. For instance, in the model described in Table 8.2 only the variables of the guest%, memory bandwidth, core stalls, and total stalls metrics are significant and therefore, their parameters are not equal to zero.

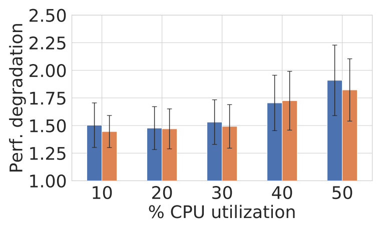
8.5 Cloud White Evaluation

8.5.1. Experimental Methodology

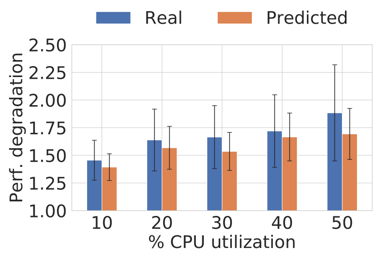
To evaluate the model accuracy, we performed a wide set of experiments both with one and multiple latency-critical applications. Firstly, we tested the effectiveness of the models by executing the target latency-critical application in a VM (*victim VM*) with two virtual CPUs (VCPUs) along with a stressor microbenchmark, launched in another VM (*inflicting VM*) running on eight VCPUs. Then, experiments were performed with multiple (two and three) latency-critical applications running in different VMs to test if Cloud White could effectively distinguish the victim and inflicting VMs and estimate the performance degradation of the victim VM. Some applications were configured to increase their load (i.e., QPS) gradually at different points of the execution time to dynamically introduce or remove interference. Each VM was launched with 2 (victim) or 4 (inflicting) VCPUs. The remaining CPUs were occupied by an instance of the microbenchmark running the *No Interference* scenario launched in a VM with 8 VCPUs.

For the experiments performed with the microbenchmark, we configured it to run the *No Interference* scenario for the first *M* seconds. During this time, Cloud White sees that the system presents a low and constant load and gathers the baseline values. Remember that the public cloud runs, most of the time, in steady phases with low CPU loads [35, 37]; therefore, baseline values would be gathered during these times. After this time, the microbenchmark adopts the stressing cache-memory-disk model (see Section 8.2.2) for *N* seconds. When this happens, Cloud White detects that a load change is taking place in the VM running the microbenchmark (i.e., acts as inflicting) and starts to apply the regression model to the VM identified as the victim to estimate its performance degradation. Finally, the microbenchmark adopts the *No Interference* model again in order to check if Cloud White can detect this new situation and estimate that little or no performance degradation is taking place. In this work, *M* and *N* are set to 60s and 50s, respectively. Similarly, in the multiple latency-critical applications experiments, we have configured inflicting applications to start increasing the load after a minimum of 45s so that Cloud White can gather the baseline values during this time.

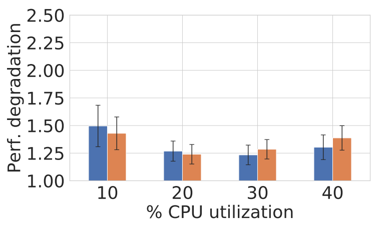
Experiments were repeated five times, point at which the deviation among the measured 95^{th} tail latencies was less than 5%, except for some experiments with higher CPU load (e.g., 40% and 50%) that experienced higher deviation between 7% and 10%. Prediction accuracy results were evaluated in terms of 1) *overall prediction error*, which defines the difference between the real and estimated performance degradation of the entire experiment, and 2) *average dynamic prediction error*, which quantifies the average error of each prediction performed with respect to the real value in each 1s quantum. In both cases, performance degradation quantifies the 95^{th} tail latency normalized against that obtained in a phase under low interference (e.g., first *M* seconds of the execution). Therefore, a value of 1.0 means no performance degradation, and higher values mean degradation has taken place.



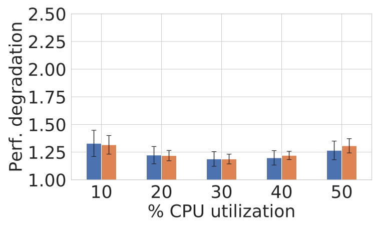
(a) `img-dnn`



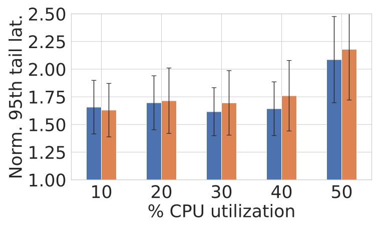
(b) `specjbb`



(c) `silo`



(d) `masstree`



(e) `moses`

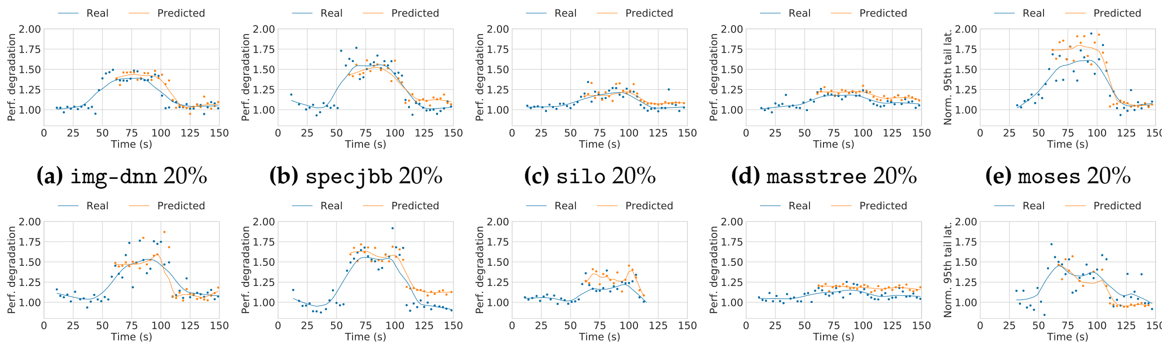
Figure 8.8: Bar plots comparing the overall real and predicted performance degradation (i.e., 95th tail latency). The CPU utilization refers to that achieved, on average, when applications were executed with *No Interference*.

8.5.2. Prediction Effectiveness with a Single Latency-Critical Application

This section shows the results obtained when applying Cloud White to the set of studied applications executed individually together with the microbenchmark. This includes those applications left outside for validation purposes (`masstree`, `moses` and `silo`), as well as those used to create and train the models (`img-dnn` and `specjbb`). The goal of including the latter two applications in the evaluation is to check if Cloud White correctly applies the appropriate model when predicting performance degradation. Notice that, in all cases, Cloud White sees these applications as black boxes and has no prior information on the applications under execution other than the resource consumption and hardware/software events gathered at run-time with the PMU.

Overall results can be observed in Figure 8.8[1]. Each graph presents the normalized real or measured performance degradation (blue bar) and the normalized estimated performance degradation (orange bar) for a variety of CPU utilization levels (X-axis), calculated as the 95th percentile of all the measurements and estimations made at run-time, respectively. Interval bars show the deviation achieved for the run-time values of both the measured and estimated normalized latency in the experiments. As it can be observed, Cloud White can estimate performance degradation accurately for most of the experiments, having, on average, a 5% prediction error. Experiments with high CPU load (i.e., 40% and 50%), however, make slightly higher prediction errors (7.5%). This is mainly because applications are close to saturating and, therefore, 95th tail latency shows more

[1] `silo` shows no results for 50% CPU utilization since the application saturates before reaching this load.



(a) `img-dnn` 20% **(b)** `specjbb` 20% **(c)** `silo` 20% **(d)** `masstree` 20% **(e)** `moses` 20%

(f) `img-dnn` 40% **(g)** `specjbb` 40% **(h)** `silo` 40% **(i)** `masstree` 40% **(j)** `moses` 40%

Figure 8.9: Dynamic graphs comparing the real and predicted performance degradation (i.e., 95^{th} tail latency).

spike values. This is also reflected in the interval bars, as longer interval bars mean that there exists a higher deviation among values. In general, we can observe that the estimated values' deviation is similar to the real values' deviation.

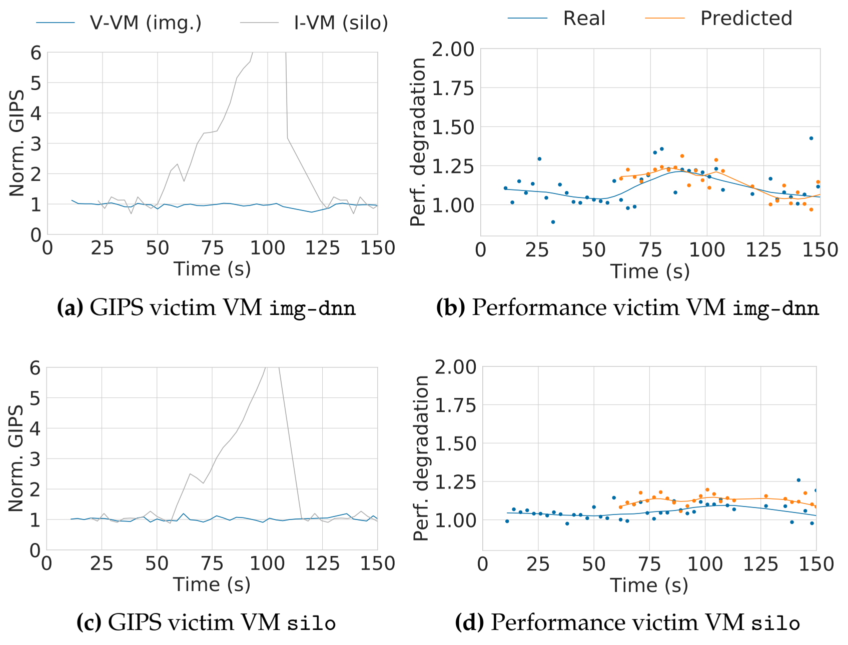
A complete evaluation, however, should also consider how Cloud White behaves when the workload changes dynamically. Figure 8.9 shows examples of how real (measured) performance degradation evolves across the execution time and the predicted performance degradation performed by the models for experiments with 20% and 40% CPU load for the first 150 seconds of execution[2]. Since 95^{th} tail latency metric experiences high variation, a best-fit curve computer with the LOWESS algorithm [162] has been plotted instead of a line joining all the points. Notice that prediction starts around the second 60, the point at which Cloud White detects that the co-running VM is changing its load and possibly causing interference. As it can be observed, both the real and predicted curves follow a similar trend in most time intervals, showing Cloud White is able to detect and predict performance degradation accurately. On average, the dynamic prediction error is less than 10%, except for some experiments like in `specjbb` with 40% CPU utilization, which is higher. Likewise, `moses` experiences a higher sensitivity to interference at the shared resources obtaining higher prediction errors in some experiments, but in all cases, the average prediction error is less than 20%.

Additionally, Cloud White is able to determine that little or no performance degradation is taking place in the last third of the execution when the co-running VM is no longer stressing the system's main resources. This proves that Cloud White can be effectively used to detect performance degradation.

8.5.3. Prediction Effectiveness with Multiple Latency-Critical Applications

This section evaluates how Cloud White behaves when running two and three VMs with latency-critical applications. For this purpose, we have chosen `img-dnn`, `silo`, and

[2]`silo` 40% has a shorter execution time since higher time resulted in QoS violation.



(a) GIPS victim VM `img-dnn`   **(b)** Performance victim VM `img-dnn`

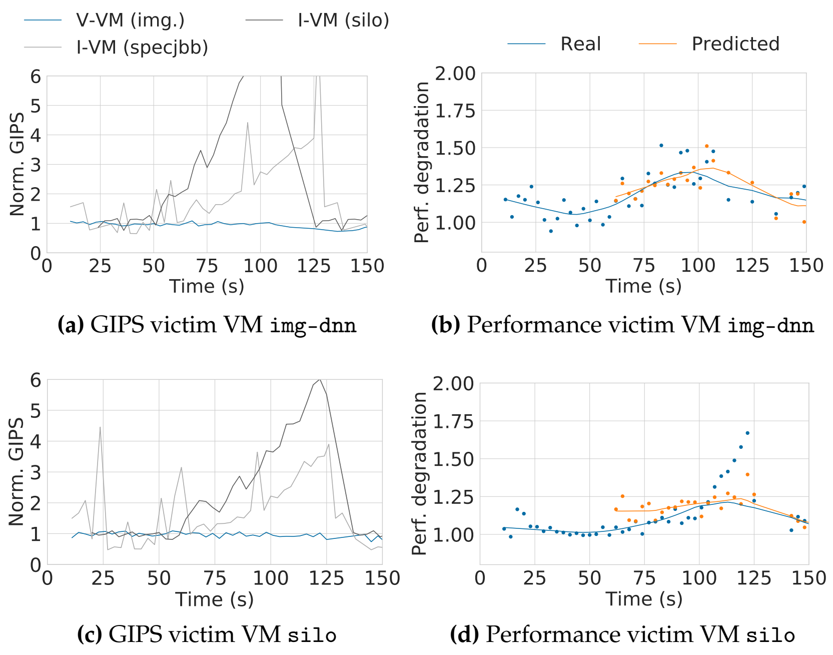
(c) GIPS victim VM `silo`   **(d)** Performance victim VM `silo`

Figure 8.10: Normalized GIPS and performance degradation (i.e., 95th tail latency) of experiments performed with 2 VMs executing `img-dnn` and `silo` as the victim (V) and inflicting (I) VMs (plots a and b) and vice-versa (plots c and d).

`specjbb`. The first two applications were chosen since they exhibited high sensitivity (`img-dnn`, see Figure 8.8a) and low sensitivity (`silo`, see Figure 8.8c) to the interference. For the three VMs experiments, `specjbb` was added with an inflicting role since it presents a significant consumption of shared resources and performs a high QPS (results shown in Chapter 7). Thus, it introduces high pressure and interference.

The main objective is to check Cloud White's two major design issues: i) distinguish the victim VM and, ii) accurately estimate its performance degradation. Although we have tested up to three VMs concurrently running latency-critical workloads, Cloud White is not limited to three. Notice, however, that in real public systems, not many latency-critical workloads are co-located in the same node due to their QoS requirements, so it is not realistic to test a high number of latency-critical workloads.

Two latency-critical applications. This section analyzes Cloud White's behavior when running two latency-critical applications. To this end, we followed the methodology (mapping of VMs to VCPUs and the way in which the interference is introduced) described in Section 8.5.1. Figure 8.10 shows the results for the first 150 seconds of two experiments performed with the pair made by `img-dnn` and `silo`; in the first experiment, *img-dnn* acts as the victim (constant 50% CPU load) and `silo` as the inflicting VM, and in the second, both applications exchange the role (`silo` runs now with constant 30% CPU load). The two upper plots (Figures 8.10a and 8.10b) show the results for the first experiment. Figure 8.10a shows the normalized GIPS with respect to the steady scenario (first 50 seconds) before `silo` starts to increase its load. As already studied in Section 8.3, the



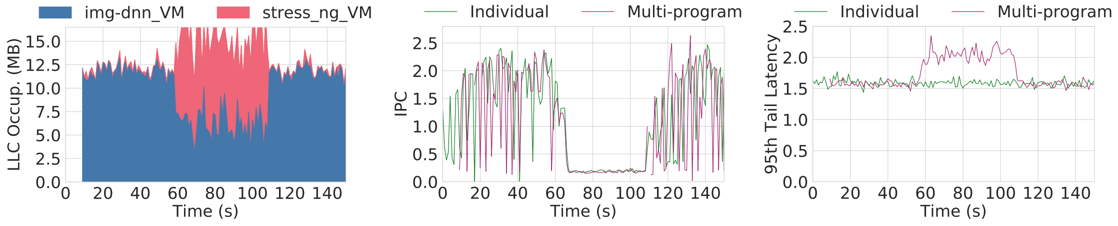
(a) GIPS victim VM `img-dnn` **(b)** Performance victim VM `img-dnn`

(c) GIPS victim VM `silo` **(d)** Performance victim VM `silo`

Figure 8.11: Normalized GIPS and performance degradation (i.e., 95th tail latency) of experiments performed with 3 VMs executing `img-dnn`, `silo` and `specjbb` as the victim (V) or inflicting (I) VMs. In plots a and b, *img-dnn* is the victim, and in plots c and d, `silo` takes this role.

GIPS of the victim VM (i.e., `img-dnn`) remain almost constant if the load experiences no variations, while the GIPS increase if the load increases, as it happens for the inflicting VM (i.e., `silo`). It can be observed that the load increase (from second 50 to second 110 approx.) matches the top of the curve of Figure 8.10b on the right side, which draws the real and predicted performance degradation of `img-dnn`, the victim VM. Around the second 60, Cloud White detects that the load of the co-running VM has increased significantly and starts to estimate the effect it has on `img-dnn`'s performance. Notice that after the second 110 approx., the load of the inflicting VM reduces, and so does the interference it introduces until it reaches the steady load again. In this experiment, Cloud White is able to estimate the performance degradation with an overall prediction error of 6% and an average dynamic prediction error of less than 10%. A similar reasoning can be applied to the two bottom figures (Figures 8.10c and 8.10d), which correspond to the second experiment where `img-dnn` is the inflicting application. In this case, the average dynamic prediction error is roughly the same, but the overall prediction error slightly lowers to 4%.

Three latency-critical applications. Finally, we analyze how Cloud White behaves with three co-located VMs running latency-critical workloads. For illustrative purposes, Figure 8.11 presents the results for `img-dnn`, `silo` and `specjbb`. In the first experiment, *img-dnn* is the victim application (constant 50% CPU load), and the other applications act as inflicting (Figures 8.11a and 8.11b). In the second experiment, `silo` acts as the victim application with a constant 20% CPU load (Figures 8.11c and 8.11d). As in the previous section, the normalized GIPS of the victim VM remains around 1 for the whole execution,



(a) LLC occupancy in multi-program **(b)** Performance of `stress_ng` in isolation **(c)** Performance of `img-dnn` in isolation

Figure 8.12: Comparison of the LLC occupancy and performance of `img-dnn` and `stress_ng`.

which allows Cloud White to identify it; meanwhile, the inflicting VMs show an increasing trend of the GIPS, matching the load increase. For the steady-load victim VM, Cloud White is able to accurately estimate the performance degradation in both cases, with an overall prediction error of 7.5% in the case where `img-dnn` is the victim VM and 8% in the case where `silo` is the victim VM and an average dynamic prediction error of 11% and 8.5%, respectively.

8.6 Comparison to Prior Work

Cloud White can detect the inter-VM interference i) considering VMs as "black boxes" while ii) assuming these VMs run latency-critical applications. Combining both statements together is one of the main contributions of this work.

This section compares our approach with the state-of-the-art approach Alita [29]. For this purpose, we implemented Alita in our experimental platform. As our proposal, this approach also handles VMs as black boxes; hence, the comparison concentrates on analyzing the online interference detection capabilities and differences between both approaches.

Alita uses low-level information, although different from that used by Cloud White. The major difference is not only in the metrics used but how the interference is detected. On the one hand, Cloud White combines numerous hardware events in a multi-variable regression model that estimates performance degradation considering resource interactions. Alita simplifies the process of estimating degradation in three main shared resources (LLC, memory bus, and CPU) in an independent way. Next, we compare the effectiveness of Cloud White against Alita from the point of view of latency-critical applications.

LLC contention. Alita evaluates LLC contention based on the *fair LLC quota* each VM should occupy, which is determined by the number of CPUs assigned to the VM. Nonetheless, as proved in Chapter 4, LLC occupancy is not a good metric to quantify contention since it does not consider data reuse. Figure 8.12 presents a counter-example to illustrate this claim, where `img-dnn` is executed together with the stressor microbenchmark (`stress_ng`), each on a VM with 8 VCPUs. `Img-dnn` shows the same behavior across all the execution while the microbenchmark occupies little LLC space (less than 1 MB) except in the second third of the execution (from 60s to 110s) where it stresses the LLC, polluting it with data with little locality (i.e., no reuse). In the first and last third of the execution, `img-dnn` occupies most of the LLC space (over 10MB out of 16.5MB). If computed the fair LLC quota, each VM should occupy 5.5MB ($8/24 \times 16.5$). Thus, Alita detects `img-dnn` is polluting the LLC when, in fact, it is not since the IPC of `stress_ng` (see Figure 8.12b) is almost the same as that achieved when running alone. On the contrary, in the second third of the execution, Alita does not detect the real LLC contention (see `img-dnn`'s tail latency increase in Figure 8.12c) since both VMs have a balanced LLC consumption. As opposed to Alita, Cloud White would properly detect this interference since i) `img-dnn` would not be detected as inflicting since its GIPS do not vary, and ii) `stress_ng` would be detected as inflicting in the middle phase where the GIPS increase.

Memory Bus contention. Alita quantifies memory bus contention by detecting *split locks* [29], which deeply degrade the performance of latency-critical applications. This approach works well on specific workloads like malicious tenant programs. We do not observe, however, split locks in the studied Tailbench applications. Moreover, when memory bandwidth contention appears in regular workloads, it is not detected by Alita, unlike Cloud White which considers memory-related metrics.

CPU contention. Contention in the CPU is quantified in terms of power. In the case of CPU-intensive applications, Alita detects CPU contention since these applications achieve a high IPC and present high circuitry activity, which significantly increases the dynamic power and temperature. On the contrary, latency-critical applications do not stress the CPU so much, and thus, we observed little increase in temperature (a few degrees Celsius at most) when running such applications.

In summary, Alita is suited to work with specific workloads presenting a high interference but cannot be considered as a general approach. In contrast, Cloud White is, to the best of our knowledge, the only approach able to detect *smooth* interference caused by inflicting VMs running latency-critical applications.

8.7 Applying Cloud White to Improve QoS

Cloud White aims to detect the inter-VM interference and quantify the performance degradation suffered by the victim VM(s). This information can be leveraged by cloud providers to carry out specific actions depending on the level of performance degradation. For instance, urgent solutions need to be carried out where a narrow margin exists with SLA violations. The most straightforward action the cloud provider can carry out in such a case is to move the identified victim VM to another less-loaded node. Notice that if this VM is not properly identified, the cloud provider cannot know which VM to move. Other more refined actions for not-so-urgent situations may be more appropriate than migration since this process is resource-intensive [176, 36]. For VMs with large image sizes, it can be very costly. These actions include analyzing the major shared processor resource that bottlenecks the performance due to interference. For instance, in case it is the LLC or memory bandwidth the resource that experiences a higher utilization increase, then the cloud provider can assign a larger share of it (e.g., LLC cache ways or memory bandwidth) to the victim VM in order to raise its performance to the same level it was before the inflicting action.

8.8 Summary

This chapter has presented an in-depth analysis of how performance degradation can be identified and quantified in a realistic scenario with multiple co-located VMs, handling VMs as black boxes and relying on metrics that can be easily monitored in the public cloud.

The proposed approach, Cloud White, uses the GIPS metric as a novel way to discern victim VM(s), which have a steady load over the last quanta. Still, their performance can degrade due to the interference introduced by the inflicting VM(s). After that, it estimates the performance degradation of the victim VM(s) using multivariable linear regression models.

Results show that Cloud White can accurately identify the victim VM(s) when running multiple VMs together. Moreover, we have shown that Cloud White can do so dynamically by detecting the co-running VMs that experience load changes. Upon a load change, Cloud White estimates the performance degradation of the victim VMs accurately.

To the best of our knowledge, this is the first approach that detects and quantifies inter-VM interference in cloud systems running latency-critical applications.

CHAPTER 9

Conclusions

This thesis has addressed the issue of resource management in server processors from two main perspectives: high-performance computing (HPC) and cloud computing. In the former, research has focused on managing two major system resources: the LLC and the processor cores. We proposed techniques efficiently distribute resources among co-running applications to improve system performance. In the latter, an extensive study has been carried out to analyze the impact of the major system resources on cloud workloads. The results are used as the basis of an online interference detection technique proposed.

This chapter outlines the main contributions of these proposals and identifies possible future directions to continue the work. Finally, the scientific publications related to this dissertation are listed.

9.1 Contributions

9.1.1. High-Performance Computing

Three main topics have been studied, focusing on the LLC and the processor cores. On the one hand, The LLC is a major shared resource in current multi-core processors. To ease the management of this resource, processor manufacturers started to include hardware tools in commercial processors that allow distributing the LLC space among cores or executing applications. In this regard, this dissertation has proposed cache partitioning management approaches that can be classified into two main topics according to the cache design: inclusive and non-inclusive LLCs. On the other hand, another major shared system resource is the processor cores. The management of this resource is critical for performance as the number of cores continues to increase in each new many-core generation. Next, these three topics are presented.

Topic 1. Partitioning inclusive LLC. Chapter 4 of this thesis has presented two cache partitioning approaches to distribute the cache space among co-running applications dynamically. Firstly, an exhaustive characterization of the behavior of SPEC CPU 2006 and

2017 applications is performed regarding the assigned LLC space and the interference at this resource. The study has analyzed the relationship between the system performance (IPC) and the LLC performance, considering the number of accesses to main memory (MPKI_LLC), reuse (HPKI_LLC), and interference with other applications (occupancy). Under these performance metrics, five cache behaviors, non-critical, sensitive, medium, squanderer, and bully, have been found. However, applications can experience multiple LLC behaviors throughout their execution which match the IPC phases of the application. The study's main contribution is that if LLC behaviors are not correctly identified, the system performance can significantly drop. Based on these findings, this dissertation proposed the Critical Aware (CA) and Critical-Phase Aware (CPA) LLC partitioning approaches. CA is a simple but effective approach considering only critical and non-critical behaviors. On the contrary, CPA considers all five behaviors and is phase-change driven, which reduces unnecessary checks on the behavior of applications and makes CPA a much more general solution.

Experimental results show that CPA improves TT over Linux default behavior by about 40% in mixes without hurting the system performance. CPA avoids performance losses CA obtains in mixes with problematic applications, improving over 20% the TT. Compared to state-of-the-art approaches, CPA outperforms KPart (on average, TT by 30%) and obtains similar TT as Dunn without sacrificing system ANTT and IPC. The main results of this research have been published in the IEEE TPDS journal and the Euro-Par 2018 conference (see J1 and C1 publications in Section 9.3).

Topic 2. Partitioning non-inclusive LLC. LLC management has also been addressed in Chapter 5 but considering the implications of a cache memory hierarchy with a non-inclusive LLC, the new memory trend in modern server processors. This new design makes the private L2 cache larger and the non-inclusive L3 cache smaller, shrinking the per-core cache space. Additionally, there is more L2-L3 traffic as evicted L2 cache blocks, whether dirty or clean, are written back to the L3 cache. These facts amplify the harmful cache pollution phenomenon; thus, cache management becomes more critical in non-inclusive caches. To tackle this issue, this thesis has proposed Cache-Poll, a cache-partitioning approach designed to effectively contain pollution and distribute space considering the application's sensitivity to the cache space. We extended the meaning of pollution to refer to any scarcely referenced blocks that replace other useful blocks in the L3 cache, identifying three main types of pollution. We used the metrics stalls caused by L2 cache misses and hits in the L3 cache (HPKI_L3) to detect those applications more sensitive to the cache space. Cache-Poll benefits from the non-inclusive L3 design by leaving little room for cache-insensitive and polluting applications, as these have fewer space requirements.

Experimental results in an Intel Skylake Xeon Silver 4116 processor with 54 12-application (as many applications as the number of cores) workloads show that Cache-Poll achieves significant performance gains over Linux OS and outperforms the state-of-the-art approach CP_{pf}, as Cache-Poll works for a high number of cache-insensitive and sensitive applications. TT gains are as high as 58% and around 24% on average. Unfairness gains are up to 127% and, on average, by 44%. The main results of this research have been published at ICPP 2022 conference (see C2 publication in Section 9.3).

Topic 3. Spatial core allocation. The current design trend in high-performance processors is to deploy many cores to match the increasing growth in the computational demand of applications, which allows launching many threads/processes to speed up computation. However, many parallel applications do not scale well with an increasing number of threads/processes due to hardware and software issues. The research has concentrated on graph applications, which, unlike other scientific applications, process vast amounts of data, which makes the default Linux time-sharing scheduler perform poorly. To overcome time-sharing weaknesses, each application should be assigned to a fraction of the processor cores, removing the intra-core interference. However, since applications present significantly irregular scalability trends with increasing numbers of cores and may exhibit different execution phases, the *optimal* number of cores can vary dynamically at execution time. This Ph.D. thesis has proposed AFAIR (Chapter 6), a core-allocation approach to co-locate multiple applications in the same server processor. AFAIR is a fair spatial core-allocation policy aimed at maximizing system utilization when executing multiple graph applications concurrently. AFAIR dynamically determines the number of cores assigned to each application by balancing the consumption of the shared memory resources. We implement a transparent method on top of the OpenMP runtime to allow applications to adapt the number of threads spawned dynamically during their execution to the number of assigned cores by AFAIR.

Experimental results show that AFAIR manages to obtain a near-optimal system fairness (94% on average). Regarding system utilization, AFAIR outperforms the Linux time-sharing scheduler and a state-of-the-art approach by reducing the total turnaround time by over 40% on average and up to 80% regardless of the number of executing applications. AFAIR can be applied directly to any commodity server processor and malleable workloads that allow dynamic adjustment. The main results of this research have been submitted to PACT 2023 conference (see R1 submission in Section 9.3).

9.1.2. Cloud Computing

Cloud systems pose additional challenges compared to typical HPC systems. The complex infrastructure of cloud systems, the nature of cloud applications, and resource efficiency are some of the major concerns with such systems.

Topic 4. Experimental platform for controlled cloud research. A major issue researchers need to face to start developing cloud research is building (or using) an experimental platform that deploys the main features of real cloud systems (types of nodes, virtualization, resource management). Existing solutions use testbeds to evaluate their work, but these platforms do not *jointly* these features and, thus, do not provide representative results. Chapter 3 presents Stratus, the experimental platform developed to carry out controlled cloud research presented in this thesis. Unlike experimental setups used in existing work, our developed experimental platform complies with all the features of cloud environments in terms of hardware and software deployment. One node of each type is included (server, client, and node), as well as a representative software stack based on a typical deployment of OpenStack. Additionally, it implements an application and resource manager that assists the researcher in the execution of experiments and resource management, which is key to designing QoS policies to mitigate inter-VM interference. Stratus has been presented at PDP 2023 conference (see C3 publication in Section 9.3).

Topic 5. Workload characterization in the public cloud. Regarding cloud applications, multi-threaded latency-critical applications represent an important subset of cloud workloads. Unlike compute-intensive applications, performance is defined in terms of latency instead of throughput since these applications must respond to the input requests within certain latency bounds to guarantee QoS (e.g., the 95^{th} or 99^{th} percentile latency) and provide a satisfactory user experience. Understanding how performance is affected by the utilization of the major system resources is a primary concern of public cloud providers. Chapter 7 characterizes the behavior, in terms of tail latency, of a set of representative latency-critical applications when varying the load (queries per second) under different conditions. Firstly, the impact of increasing the number of threads and different thread-to-core allocation policies (single-task and multi-task) are analyzed, revealing that some applications' performance does not scale and shows a Hyper-Threading insensitive behavior. To identify these applications at run-time to improve resource utilization, an analysis of the major system resource consumption was carried out, concluding that applications with higher resource requirements can be identified by checking the CPU utilization and memory bandwidth trend with an increasing number of threads. The previous analysis is performed considering the applications running alone in the system. However, cloud systems collocate multiple applications in the same server node, so they must share the main system resource. To assess the performance in this situation, we analyze the impact of reducing the LLC, memory bandwidth, and disk bandwidth using resource partitioning tools available in commercial processors. Results show that enough resource share should be provided to sustain performance, especially the memory and disk bandwidth which correlate strongly with the performance. The main results of this research have been published FGCS journal (see J2 publication in Section 9.3).

Topic 5. Inter-VM interference in the public cloud. To avoid QoS violations due to inter-VM interference, cloud providers typically adopt an overprovisioning strategy, assigning resources in excess to each VM to prevent degradation. However, this strategy results in poor resource utilization. Accurately estimating performance degradation due to inter-VM interference would allow cloud providers to improve resource utilization while meeting applications' QoS requirements. Chapter 8 proposes Cloud White, an approach that detects and accurately estimates inter-VM interference in scenarios with multiple co-located latency-critical VMs. This is especially challenging since the performance of latency-critical is defined by the tail latency (e.g., 95^{th} percentile), which has a non-linear behavior and is more sensitive to interference. Furthermore, VMs are treated as black boxes, meaning no information about the running VMs is provided. With Cloud White, the VM is not a black box anymore, but its behavior (and introduced interference) is revealed, becoming "white" or Cloud White. Cloud White works in two main stages. Firstly, the victim and inflicting VM(s) are discerned, considering whether or not the load is steady using the GIPS metric. After that, it estimates the performance degradation of the victim VM(s) using multi-variable linear regression models. Experimental results show that Cloud White is able to identify the victim VM(s) in scenarios of multiple running VMs and estimate the performance degradation they experience, with a total error deviation of about 5% and an average dynamic prediction error of less than 10%. To the best of our knowledge, this is the first approach that detects and quantifies inter-VM interference in cloud systems running latency-critical applications. Cloud White approach has been published in FGCS journal (see J3 publication in Section 9.3).

9.2 Future Directions

9.2.1. Evolving Systems and Applications

The current trend in server processors found in data centers is implementing multi-socket NUMA nodes with increasing cores. This opens a new research spectrum as a higher number of cores allows the execution of a higher number of *critical* high resource-demanding applications. In this regard, emerging workloads, such as big data, artificial intelligence, and graph, are becoming more prevalent in current data centers.

The fact that NUMA nodes comprise multiple sockets implies an additional decision-making step for the resource manager, as applications must first be scheduled to one of the sockets. Then, for each socket, resources are distributed among the scheduled applications.

9.2.2. Future Research Topics

Research needs to be defined considering emerging workloads, application domains, and the processor and memory system trend.

Regarding LLC partitioning, the research can address approaches in multiple sockets and nodes, taking into account both main memory bandwidth restrictions and prefetch gains considering the overall workload. Notice that the systems described above will impose new challenges as new latencies (e.g., NUMA), memory constraints, and a higher number of *critical* applications must be considered. Regarding core allocation, systems with more cores will result in higher power consumption and heat dissipation. Therefore, resource management strategies will need to take into consideration energy efficiency in addition to system performance.

Concerning cloud computing, dynamic resource provisioning is currently a hot research topic. The information provided by the proposed prediction model in this dissertation, Cloud White, can be leveraged by a resource manager to take corrective actions. That is, Cloud White can be used to detect VMs suffering low performance due to inter-VM interference. Then, a resource manager can be designed to analyze the resources that bottleneck the performance and partition them accordingly (e.g., limit LLC to the inflicting VMs).

The approaches proposed in this thesis are performed using heuristics and multi-variable regression models. Although promising experimental results are obtained, these approaches require an arduous experimental process to establish thresholds or build predictive models when new conditions are encountered (e.g., applications with an *unknown* behavior). To address these weaknesses, as for future work, we plan to implement machine learning methods into Stratus to build learning models. Applying machine learning would allow thresholds or prediction models to be dynamically adjusted and learn from data collected in the running experiments.

9.3 Publications

Below are listed the works exclusively related to this thesis. The contributions of the Ph.D. candidate to the works are the design and implementation of the proposed approaches, conducting and analyzing the experiments, writing the paper drafts, and presenting the papers at the conferences. The co-authors of these works have collaborated in designing the proposals, analyzing the experimental results, and writing the papers. Their valuable advice has enriched the work carried out in this Ph.D. Thesis.

9.3.1. Published Papers

The following papers were submitted and accepted in international journals, international conferences, and domestic conferences.

International Journals

J1. Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, Julio Pons. "Phase-Aware Cache Partitioning to Target Both Turnaround Time and System Performance". In *IEEE Transactions on Parallel and Distributed Systems* (Q2), vol. 31, no. 1, pp. 2556-2568, 2020.

J2. Lucia Pons, Josué Feliu, José Puche, Chaoyi Huang, Salvador Petit, Julio Pons, María E. Gómez, Julio Sahuquillo. "Effect of Hyper-Threading in Latency-Critical Multithreaded Cloud Applications and Utilization Analysis of the Major System Resources". In *Future Generation Computer Systems* (Q1), vol. 131, pp. 194-208, 2022.

J3. Lucia Pons, Josué Feliu, Julio Sahuquillo, María E. Gómez, Salvador Petit, Julio Pons, Chaoyi Huang. "Cloud White: Detecting and Estimating QoS Degradation of Latency-Critical Workloads in the Public Cloud". In *Future Generation Computer Systems* (Q1), vol. 138, pp. 13-25, 2023.

International Conferences

C1. Lucia Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, Julio Pons. "Improving System Turnaround Time with Intel CAT by Identifying LLC Critical Applications". In *Proceedings of the 24th International European Conference on Parallel and Distributed Computing (Euro-Par)*, pp. 603-615, Turin, Italy, 2018. CORE A.

C2. Lucia Pons, Julio Sahuquillo, Salvador Petit, Julio Pons. "Cache-Poll: Containing Pollution in Non-Inclusive Caches Through Cache Partitioning". In *Proceedings of the*

51st International Conference on Parallel Processing (ICPP), pp. 33:1-33:11, virtual, 2022. CORE A.

C3. Lucia Pons, Salvador Petit, Julio Pons, María E. Gómez, Chaoyi Huang and Julio Sahuquillo. "Stratus: A Hardware/Software Infrastructure for Controlled Cloud Research". In *Proceedings of 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 299-306, Naples, Italy, 2023. This publication received the Best Paper Award in the Special Session "Cloud Computing on Infrastructure as a Service and its Applications". CORE C.

Domestic Conferences and Posters

- Lucía Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, María E. Gómez, Julio Pons. "The memory wall in multicore processors: mitigating the LLC interference with Intel CAT". Poster presented in *Informática para Tod@s*, Madrid, Spain, 2018.
- Lucía Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, Julio Pons. "Mejora del Turnaround Time con la Tecnología CAT de Intel". In *Actas de las XXIX Jornadas de Paralelismo*, pp. 153-159, Teruel, Spain, 2018.
- Lucía Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, Julio Pons. "Criticality Aware LLC Partitioning: Reducing System Turnaround Time with Intel CAT". Poster presented in *Informática para Tod@s*, A Coruña, Spain, 2019.
- Lucía Pons, Josué Feliu, Salvador Petit, Julio Pons, María E. Gómez y Julio Sahuquillo. "Caracterización de las aplicaciones de latencia crítica Tailbench en un entorno cloud". In *Actas de las XXXI Jornadas de Paralelismo*, pp. 503-512, Málaga, Spain, 2021.
- Lucia Pons, Josué Feliu, Chaoyi Huang, Salvador Petit, Julio Pons, María E. Gómez, Julio Sahuquillo. "Exploiting Hyper-Threading in the Public Cloud". In *Proceedings of the 18th International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems (ACACES)*, pp. 147-150, Fiuggi, Italy, 2022.
- Lucía Pons, Julio Sahuquillo, Salvador Petit, Julio Pons. "Caracterización de las prestaciones y la polución en las caches no inclusivas". In *Actas de las XXXII Jornadas de Paralelismo*, pp. 345-351, Alicante, Spain, 2022.

9.3.2. Papers Under Review

The following paper was submitted and is currently under review:

R1. Lucia Pons, Julio Sahuquillo, and Timothy M. Jones. "AFAIR: A Fair Spatial Scheduler for Accelerating Graph Workloads on Commodity Servers". Submitted to *32nd International Conference on Parallel Architectures and Compilation Techniques (PACT 2023)*. CORE B.

The research skills acquired during the development of this thesis have helped to propose a new approach that aids Computer Architecture Instructors when teaching SMT concepts. In this regard, we propose teaching SMT topics in real hardware under machine-learning workloads. The approach, as well as the illustrative experimental results, have been submitted to the Elsevier Advance in Computers journal:

- Lucia Pons, Marta Navarro, Salvador Petit, Julio Pons, María E. Gómez, and Julio Sahuquillo. "SMT Efficiency in Supervised ML Methods: A Throughput and Interference Analysis". Submitted to *Elsevier Advances in Computers* journal (Q2).

The results of this work have not been included as part of this manuscript as we preferred to keep the manuscript focused on research proposals.

9.3.3. Unsuccessful Attempts

Failure is inevitable in growth and essential to success, especially when pursuing a research career. The valuable feedback provided by the reviewers has contributed to improving the quality of the work and helped me develop essential skills in research, such as self-improvement and self-criticism.

To conclude, we would like to list the rejected submitted papers:

- Lucia Pons, Josué Feliu, José Puche, Chaoyi Huang, Salvador Petit, Julio Pons, María E. Gómez, Julio Sahuquillo. "Understanding Cloud Workloads Performance in a Production like Environment". Submitted to *Journal of Parallel and Distributed Computing*, October 2020.
- Lucia Pons, Josué Feliu, Julio Sahuquillo, María E. Gómez, Salvador Petit, Julio Pons, Huang Chaoyi. "Cloud White: A Production System Approach for Estimating QoS Degradation in the Public Cloud", submitted to the *28th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2022).*
- Lucia Pons, Josué Feliu, Julio Sahuquillo, María E. Gómez, Salvador Petit, Julio Pons, Huang Chaoyi. "Cloud White: A Production System Approach for Estimating QoS Degradation in the Public Cloud", submitted to the *49th International Symposium on Computer Architecture (ISCA 2022).*
- Lucia Pons, Julio Sahuquillo, Salvador Petit, Julio Pons. "Cache-Poll: Containing Pollution in Non-Inclusive Caches Through Cache Partitioning". Submitted to *ACM International Conference on Supercomputing (ICS 2022).*
- Lucia Pons, Julio Sahuquillo, Timothy Jones. "GRASS: Accelerating Graph Workloads on Commodity Servers through Spatial Scheduling", submitted to the *50th International Symposium on Computer Architecture (ISCA 2023).*

References

[1] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology," in *Procedings of PACT*, 2017, pp. 194–205.

[2] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of ISCA*, 1990, pp. 364–373.

[3] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer, "PACMan: Prefetch-Aware Cache Management for high performance caching," in *Proceedings of MICRO*, 2011, pp. 442–453.

[4] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 Processor Architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, Mar 2017.

[5] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.

[6] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: A fundamental relook at multi-level cache hierarchies," in *Proceedings of ISCA*. IEEE, 2018, pp. 96–109.

[7] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín, and V. Viñals-Yúfera, "Memory hierarchy characterization of SPEC CPU2006 and SPEC CPU2017 on the Intel Xeon Skylake-SP," *PLOS ONE*, vol. 14, no. 8, pp. 1–24, 08 2019.

[8] Arm Developer, "Arm DynamIQ Shared Unit-110 Technical Reference Manual - L3 cache [Online]," Available: https://developer.arm.com/documentation/102639/0201/L3-cache, 2021, accessed: 2022-02-01.

[9] Advanced Micro Devices, Inc., "AMD64 Technology Platform Quality of Service Extensions [Online]," Available: https://www.amd.com/system/files/TechDocs/56375_1.03_PUB.pdf, 2022, accessed: 2023-04-19.

[10] Intel Corporation, "Improving Real-Time Performance by Utilizing Cache Allocation Technology. White Paper: 31843-001US," Available: http://www.intel.com/content/dam/www/public/us/en/documents/

white-papers/cache-allocation-technology-white-paper.pdf, 2015, accessed: 2022-02-01.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of ISCA*, 1995, pp. 392–403.

[12] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, 2002.

[13] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Addressing fairness in smt multicores with a progress-aware scheduler," in *Proceedings of IPDPS*, 2015, pp. 187–196.

[14] Chris Goetting, "Intel 4th Gen Xeon Scalable Sapphire Rapids Performance Review [Online]," Available: https://hothardware.com/reviews/intel-4th-gen-xeon-scalable-performance-review, 2023, accessed: 2023-04-25.

[15] Advanced Micro Devices, Inc, "4th Gen AMD EPYC Processor Architecture. White Paper." Available: https://www.amd.com/en/campaigns/epyc-9004-architecture, 2022, accessed: 2023-04-25.

[16] 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 *et al.*, "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of PACT*, 2008, pp. 72–81.

[18] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[19] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," *Proceedings of ASPLOS*, pp. 37–48, 2012.

[20] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs," *ACM Sigplan Notices*, vol. 43, no. 3, pp. 277–286, 2008.

[21] A. F. Lorenzon, C. C. De Oliveira, J. D. Souza, and A. C. S. Beck, "Aurora: Seamless Optimization of OpenMP Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 5, pp. 1007–1021, 2018.

[22] V. S. da Silva, A. G. Nogueira, E. C. de Lima, H. M. de A. Rocha, M. S. Serpa, M. C. Luizelli, F. D. Rossi, P. O. Navaux, A. C. S. Beck, and A. Francisco Lorenzon, "Smart resource allocation of concurrent execution of parallel applications," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, pp. 1–15, 2021.

[23] "Amazon's EC2 [Online]," Available: http://aws.amazon.com/ec2/, 2020, accessed: 2020-09-30.

[24] "Windows Azure [Online]," Available: http://www.windowsazure.com/, 2020, accessed: 2020-09-30.

[25] "Google Compute Engine [Online]," Available: https://developers.google.com/compute/, 2020, accessed: 2020-09-30.

[26] J. R. Hamilton, "Cost of Power in Large-Scale Data Centers [Online]," Available: https://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/, 2020, accessed: 2020-09-27.

[27] A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B.-S. Lee, P. Faraboschi, R. Kaufmann, and D. Milojicic, "The Who, What, Why, and How of High Performance Computing in the Cloud," in *Proceedings of CloudCom*, 2013, pp. 306–314.

[28] A. Suresh and A. Gandhi, "ServerMore: Opportunistic Execution of Serverless Functions in the Cloud," in *Proceedings of SoCC*, 2021, pp. 570–584.

[29] Q. Chen, S. Xue, S. Zhao, S. Chen, Y. Wu, Y. Xu, Z. Song, T. Ma, Y. Yang, and M. Guo, "Alita: Comprehensive Performance Isolation through Bias Resource Management for Public Clouds," in *Proceedings of SC*, 2020, pp. 1–13.

[30] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in *Proceedings of HPCA*, 2020, pp. 193–206.

[31] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of ASPLOS*, 2019, pp. 107–120.

[32] S. A. Javadi, A. Suresh, M. Wajahat, and A. Gandhi, "Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments," in *Proceedings of SoCC*, 2019, pp. 272–285.

[33] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications," in *Proceedings of CLOUD*, 2018, pp. 82–89.

[34] Y. Sfakianakis, M. Marazakis, and A. Bilas, "Skynet: Performance-driven Resource Management for Dynamic Workloads," in *Proceedings of CLOUD*, 2021, pp. 527–539.

[35] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on Alibaba cluster trace," in *Proceedigs of IEEE Big Data*, 2017, pp. 2884–2892.

[36] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms," in *Proceedings of SOSP*, 2017, pp. 153–167.

[37] Q. Liu and Z. Yu, "The Elasticity and Plasticity in Semi-Containerized Co-Locating Cloud Workload: A View from Alibaba Trace," in *Proceedings of SoCC*, 2018, pp. 347–360.

[38] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of MICRO*, 2006, pp. 423–432.

[39] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *Proceedings of MICRO*, 2015, pp. 62–75.

[40] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-grain Cache Partitioning," in *Proceedings of ISCA*, 2011, pp. 57–68.

[41] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic Shared Cache Management (PriSM)," in *Proceedings ISCA*, 2012, pp. 428–439.

[42] J. Sahuquillo and A. Pont, "The filter cache: A run-time cache management approach," in *Proceedings of EUROMICRO*, vol. 1. IEEE, 1999, pp. 424–431.

[43] F. Shen, Y. He, J. Zhang, Q. Li, J. Li, and C. Xu, "Reuse locality aware cache partitioning for last-level cache," *Computers & Electrical Engineering*, vol. 74, pp. 319–330, 2019.

[44] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu, "Rethinking memory management in modern operating system: Horizontal, vertical or random?" *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1921–1935, 2015.

[45] J. Park, H. Yeom, and Y. Son, "Page reusability-based cache partitioning for multicore systems," *IEEE Transactions on Computers*, vol. 69, no. 6, pp. 812–818, 2020.

[46] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Procedings of HPCA*, 2018, pp. 104–117.

[47] K. Nikas, N. Papadopoulou, D. Giantsidi, V. Karakostas, G. Goumas, and N. Koziris, "DICER: Diligent Cache Partitioning for Efficient Workload Consolidation," in *Proceedings of ICPP*, 2019, pp. 1–10.

[48] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic Cache Allocation with Partial Sharing," in *Proceedings of EuroSys*, 2018, pp. 1–15.

[49] G. Sun, J. Shen, and A. V. Veidenbaum, "Combining prefetch control and cache partitioning to improve multicore performance," in *Proceedings of IPDPS*, 2019, pp. 953–962.

[50] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of ISCA*, 2015, pp. 450–462.

[51] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-Driven LLC Allocation," in *Proceedings of USENIX*, 2016, pp. 295–308.

[52] Y. Kim, A. More, E. Shriver, and T. Rosing, "Application Performance Prediction and Optimization Under Cache Allocation Technology," in *Proceedings of DATE*, 2019, pp. 1285–1288.

[53] J. Xiao, A. Pimentel, and X. Liu, "CPpf a prefetch aware LLC partitioning approach," in *Proceedings of ICPP*, 08 2019, pp. 1–10.

[54] J. Park, S. Park, and W. Baek, "CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers," in *Proceedings of EuroSys*, 2019, pp. 1–16.

[55] J. C. Saez, F. Castro, G. Fanizzi, and M. Prieto-Matias, "LFOC+: A Fair OS-level Cache-Clustering Policy for Commodity Multicore Systems," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1952–1967, 2021.

[56] B. Chatterjee, S. Khan, and S. Pande, "Com-CAS: Effective Cache Apportioning under Compiler Guidance," in *Proceedings of PACT*, 2022, pp. 14–27.

[57] W. Tang, S. Fu, Y. Ke, Q. Peng, and F. Gao, "Themis: Fair Memory Subsystem Resource Sharing with Differentiated QoS in Public Clouds," in *Proceedings of ICPP*, 2022, pp. 1–12.

[58] S. Kundan, T. Marinakis, I. Anagnostopoulos, and D. Kagaris, "A Pressure-Aware Policy for Contention Minimization on Multicore Systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–26, 2022.

[59] G. Galante and R. da Rosa Righi, "Adaptive parallel applications: from shared memory architectures to fog computing (2002–2022)," *Cluster Computing*, vol. 25, no. 6, pp. 4439–4461, 2022.

[60] R. Sudarsan and C. J. Ribbens, "Combining performance and priority for scheduling resizable parallel applications," *Journal of Parallel and Distributed Computing*, vol. 87, pp. 55–66, 2016.

[61] D. Yang, W. Rang, D. Cheng, Y. Wang, J. Tian, and D. Tao, "Elastic Executor Provisioning for Iterative Workloads on Apache Spark," in *Proceedings of Big Data*, 2019, pp. 413–422.

[62] G. Georgakoudis, H. Vandierendonck, P. Thoman, B. R. D. Supinski, T. Fahringer, and D. S. Nikolopoulos, "SCALO: Scalability-Aware Parallelism Orchestration for Multi-Threaded Workloads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–25, 2017.

[63] S. M. V. Marques, M. S. Serpa, A. N. Muñoz, F. D. Rossi, M. C. Luizelli, P. O. Navaux, A. C. S. Beck, and A. F. Lorenzon, "Optimizing the EDP of OpenMP applications via concurrency throttling and frequency boosting," *Journal of Systems Architecture*, vol. 123, pp. 1–12, 2022.

[64] Y. Cho, C. A. C. Guzman, and B. Egger, "Maximizing system utilization via parallelism management for co-located parallel applications," in *Proceedings of PACT*, 2018, pp. 1–14.

[65] S. Srikanthan, S. Chakraborti, P. Ferro, and S. Dwarkadas, "MAPPER: Managing Application Performance via Parallel Efficiency Regulation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 2, pp. 1–26, 2022.

[66] M. K. Moori, H.-M. G. de A. Rocha, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Improving the Efficiency of graph algorithm executions on high-performance computing," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, pp. 1–17, 2022.

[67] H. M. G. de A. Rocha, J. Schwarzrock, A. F. Lorenzon, and A. C. S. Beck, "Using Machine Learning to Optimize Graph Execution on NUMA Machines," in *Proceedings of DAC*, 2022, pp. 1027–1032.

[68] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *Proceedings of MICRO*, 2015, pp. 178–190.

[69] S. Ainsworth and T. M. Jones, "Graph Prefetching Using Data Structure Knowledge," in *Proceedings of ICS*, 2016, pp. 1–16.

[70] J. Feliu, A. Naithani, J. Sahuquillo, S. Petit, M. Qureshi, and L. Eeckhout, "VMT: Virtualized Multi-Threading for Accelerating Graph Workloads on Commodity Processors," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1386–1398, 2022.

[71] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads," in *Proceedings of HPCA*, 2019, pp. 373–386.

[72] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of ISCA*, 2015, pp. 105––117.

[73] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of MICRO*, 2016, pp. 1–13.

[74] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "TuNao: A High-Performance and Energy-Efficient Reconfigurable Accelerator for Graph Processing," in *Proceedings of CCGRID*, 2017, pp. 731–734.

[75] J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang, and H. Yu, "TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing," in *Proceedings of ISCA*, 2022, pp. 116–129.

[76] S. Eyerman, W. Heirman, K. Du Bois, J. B. Fryman, and I. Hur, "Many-Core Graph Workload Analysis," in *Proceedings of SC*, 2018, pp. 282–292.

[77] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, and S. Katti, "Parallel Graph Processing: Prejudice and State of the Art," in *Proceedings of ICPE*, 2016, pp. 85–90.

[78] S. Beamer, K. Asanovic, and D. Patterson, “Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server,” in *Proceedings of IISWC*, 2015, pp. 56–65.

[79] A. Samara and J. Tuck, “The Case for Domain-Specialized Branch Predictors for Graph-Processing,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 101–104, 2020.

[80] O. Green, M. Dukhan, and R. Vuduc, “Branch-Avoiding Graph Algorithms,” in *Proceedings of SPAA*, 2015, pp. 212–223.

[81] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti, “Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges,” in *Proceedings of MASCOTS*, 2016, pp. 49–58.

[82] L. Jiang, L. Chen, and J. Qiu, “Performance Characterization of Multi-threaded Graph Processing Applications on Many-Integrated-Core Architecture,” in *Proceedings of ISPASS*, 2018, pp. 199–208.

[83] A. Limaye, A. Tumeo, and T. Adegbija, “Energy characterization of graph workloads,” *Sustainable Computing: Informatics and Systems*, vol. 29, pp. 1–10, 2021.

[84] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, “Hypart: A Hybrid Technique for Practical Memory Bandwidth Partitioning on Commodity Servers,” in *Proceedings of PACT*, 2018, pp. 1–14.

[85] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack amp; Cap: Adaptive DVFS and thread packing under power caps,” in *Proceedings of MICRO*, 2011, pp. 175–185.

[86] R. Schöne, T. Ilsche, M. Bielert, D. Molka, and D. Hackenberg, “Software Controlled Clock Modulation for Energy Efficiency Optimization on Intel Processors,” in *Proceedings of E2SC*, 2016, pp. 69–76.

[87] A. Jeatsa, B. Teabe, and D. Hagimont, “CASY: A CPU Cache Allocation System for FaaS Platform,” in *Proceedings of CCGrid*, 2022, pp. 494–503.

[88] C. Delimitrou and C. Kozyrakis, “Quasar: Resource-Efficient and QoS-Aware Cluster Management,” in *Proceedings of ASPLOS*, vol. 49, 2014, pp. 127–144.

[89] ——, “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems,” in *SIGARCH Comput. Archit. News*, 2016, pp. 473–488.

[90] G. Torres and C. Liu, “Adaptive Virtual Machine Management in the Cloud: A Performance-Counter-Driven Approach,” *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, vol. 4, no. 2, pp. 28–43, 2014.

[91] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, “DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments,” in *Proceedings of USENIX ATC*, 2013, pp. 219–230.

[92] N. Vasic, D. M. Novakovic, S. Miucin, D. Kostic, and R. Bianchini, "DejaVu: accelerating resource allocation in virtualized environments," in *Proceedings of ASPLOS*, 2012, pp. 423–436.

[93] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads," in *Proceedings of HPCA*, 2017, pp. 409–420.

[94] H. Moradi, W. Wang, and D. Zhu, "Online Performance Modeling and Prediction for Single-VM Applications in Multi-Tenant Clouds," *IEEE Transactions on Cloud Computing*, vol. 11, no. 1, pp. 97–110, 2021.

[95] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proceedings of ISCA*, 2013, pp. 607–618.

[96] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for QoS-aware clouds," in *Proceedings of EuroSys*, Apr. 2010, pp. 237–250.

[97] M. Melo Alves and L. M. d. A. Drummond, "A Multivariate and Quantitative Model for Predicting Cross-Application Interference in Virtual Environments," *Journal of Systems and Software*, vol. 128, no. C, pp. 150–163, 2017.

[98] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2020.

[99] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in *Proceedings of HPCA*, 2020, pp. 167–179.

[100] P. Alcorn, "Intel Xeon Platinum 8176 Scalable Processor Review [Online]," Available: https://www.tomshardware.com/reviews/intel-xeon-platinum-8176-scalable-cpu,5120-4.html, 2023, accessed: 2023-02-20.

[101] A. Kumar and M. Trivedi, "Intel Xeon Scalable Processor Architecture Deep Dive," 2017, presentation at Intel Press Workshops.

[102] "Google Cloud Compute Engine - CPU platforms [Online]," Available: https://cloud.google.com/compute/docs/cpu-platforms, 2022, accessed: 2022-11-14.

[103] "Huawei Elastic Cloud Server (ECS) [Online]," Available: https://www.huaweicloud.com/intl/en-us/product/ecs.html, 2022, accessed: 2022-11-14.

[104] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of OSDI*, 2006, pp. 307–320.

[105] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an Open-source Solution for Cloud Computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, Mar. 2012.

[106] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8, 2007, pp. 225–230.

[107] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[108] "Amazon Web Services [Online]," Available: https://aws.amazon.com/ec2/faqs/?nc1=h_ls, 2022, accessed: 2022-11-28.

[109] "Google Compute Engine FAQ [Online]," Available: https://cloud.google.com/compute/docs/faq, 2022, accessed: 2022-11-28.

[110] "libvirt: The virtualization API [Online]," Available: https://libvirt.org, 2022, accessed: 2022-11-28.

[111] "QEMU [Online]," Available: https://www.qemu.org, 2022, accessed: 2022-11-28.

[112] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proceedings of NSDI*, May 2015, pp. 117–130.

[113] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[114] "DPDK [Online]," Available: https://www.dpdk.org/, 2022, accessed: 2022-11-28.

[115] A. C. de Melo, "Performance counters on Linux," in *Linux Plumbers Conference*, vol. 118, 2009.

[116] B. Cai, K. Li, L. Zhao, and R. Zhang, "Less Provisioning: A Hybrid Resource Scaling Engine for Long-Running Services With Tail Latency Guarantees," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1941–1957, 2022.

[117] Andrew H., Abbasi, Khawar M., Marcel C., "Introduction to Memory Bandwidth Allocation," Available at https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-allocation, 2019.

[118] Michael Bayer et al., "Mako Templates [Online]," Available: http://www.makotemplates.org/, 2019, accessed: 2022-11-29.

[119] D. L. Mills, "Internet time synchronization: the network time protocol," *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[120] S. Chen, A. Jin, C. Delimitrou, and J. F. Martínez, "ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud," in *Proceedings of HPCA*, 2022, pp. 155–168.

[121] R. Jia, Y. Yang, J. Grundy, J. Keung, and L. Hao, "A systematic review of scheduling approaches on multi-tenancy cloud platforms," *Information and Software Technology*, vol. 132, pp. 1–16, 2021.

[122] Z. Wang, C. Xu, K. Agrawal, and J. Li, "Adaptive scheduling of multiprogrammed dynamic-multithreading applications," *Journal of Parallel and Distributed Computing*, vol. 162, pp. 76–88, 2022.

[123] *Standard Performance Evaluation Corporation, SPEC CPU 2006 [Online]*, Available: http://spec.org/cpu2006, accessed: 2018-03-30.

[124] *Standard Performance Evaluation Corporation, SPEC CPU 2017 [Online]*, Available: http://spec.org/cpu2016, accessed: 2018-03-30.

[125] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of IISWC*, 2016, pp. 1–10.

[126] R. Baeza-Yates, "Applications of web query mining," in *Proceedings of ECIR*, 2005, pp. 7–22.

[127] D. G. Feitelson, *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[128] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[129] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *Proceedings of ISPASS*, 2001, pp. 164–165.

[130] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proceedings of PACT*, 2013, pp. 177–187.

[131] S. Eyerman and L. Eeckhout, "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 93–96, 2014.

[132] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems," in *Proceedings of ASPLOS*, 2016, pp. 33–47.

[133] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of MICRO*, Dec 2003, pp. 217–227.

[134] G. L. T. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, and G. da Costa, "A User Friendly Phase Detection Methodology for HPC Systems' Analysis," in *Proceedings of GREENCOM-ITHINGS-CPSCOM*, 2013, pp. 118–125.

[135] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient software-based online phase classification," in *Proceedings of IISWC*, Nov 2011, pp. 104–115.

[136] X. Liao, R. Guo, D. Yu, H. Jin, and L. Lin, "A Phase Behavior Aware Dynamic Cache Partitioning Scheme for CMPs," *International Journal of Parallel Programming*, vol. 44, pp. 68–86, 02 2016.

[137] J. Miller, "Short Report: Reaction Time Analysis with Outlier Exclusion: Bias Varies with Sample Size," *Journal of Experimental Psychology*, vol. 43, no. 4, pp. 907–912, 1991.

[138] J. Hofmann, C. L. Alappat, G. Hager, D. Fey, and G. Wellein, "Bridging the architecture gap: abstracting performance-relevant properties of modern server processors," *arXiv preprint arXiv:1907.00048*, 2019.

[139] T. Gruber and J. Hammer, "L2 L3 MEM traffic on Intel Skylake SP CascadeLake SP," Available at https://github.com/RRZE-HPC/likwid/wiki/L2-L3-MEM-traffic-on-Intel-Skylake-SP-CascadeLake-SP, 2019, accessed: 2022-04-20.

[140] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *Proceedings of ISPASS*, 2014, pp. 35–44.

[141] J. D. McCalpin, "HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor," in *Proceedings of SC18*, 2018, pp. 225–237.

[142] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores," in *Proceedings of PACT*, 2013, pp. 177–188.

[143] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One Trillion Edges: Graph Processing at Facebook-Scale," in *Proceedings of VLDB Endowment*, 2015, pp. 1804–1815.

[144] L. Quick, P. Wilkinson, and D. Hardcastle, "Using Pregel-like Large Scale Graph Processing Frameworks for Social Network Analysis," in *Proceedings of ASONAM*, 2012, pp. 457–463.

[145] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: survey and an experimental evaluation," *Cluster Computing*, vol. 18, no. 3, pp. 1189–1213, 2015.

[146] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux^2: Distributed Graph Computation for Machine Learning," in *Proceedings of NSDI*, 2017, pp. 669–682.

[147] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph Convolutional Neural Networks for Web-Scale Recommender Systems," in *Proceedings of KDD*, 2018, pp. 974–983.

[148] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of PPoPP*, 2013, pp. 135–146.

[149] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-Centric Graph Processing Using Streaming Partitions," in *Proceedings of SOSP*, 2013, pp. 472–488.

[150] S. Grossman, H. Litz, and C. Kozyrakis, "Making Pull-Based Graph Processing Performant," in *Proceedings of PPoPP*, 2018, pp. 246–260.

[151] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at What Cost?" in *Proceedings of HOTOS*, 2015, p. 14.

[152] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable," *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 1, pp. 1–70, 2021.

[153] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.

[154] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[155] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, and S. Bagchi, "Pythia: Improving Datacenter Utilization via Precise Contention Prediction for Multiple Co-Located Workloads," in *Proceedings of Middleware*, 2018, pp. 146–160.

[156] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *Proceedings of SOSP*, 2011, pp. 143–157.

[157] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[158] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, vol. 61, no. 8, pp. 65–72, 2018.

[159] D. A. Menasce, "QoS issues in web services," *IEEE internet computing*, vol. 6, no. 6, pp. 72–75, 2002.

[160] F. Raimondi, J. Skene, and W. Emmerich, "Efficient online monitoring of web-service SLAs," in *Proceedings of SIGSOFT*, 2008, pp. 170–180.

[161] W. S. Cleveland, "Robust locally weighted regression and smoothing scatterplots," *Journal of the American statistical association*, vol. 74, no. 368, pp. 829–836, 1979.

[162] J. D. Triveri, "LOESS - Nonparametric Scatterplot Smoothing in Python [Online]," Available: http://www.jtrive.com/loess-nonparametric-scatterplot-smoothing-in-python.html, 2018, accessed: 2020-12-21.

[163] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of EuroSys*, 2012, pp. 183–196.

[164] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of SOSP*, 2013, pp. 18–32.

[165] J. R. Bulpin and I. Pratt, "Hyper-Threading Aware Process Scheduling Heuristics," in *Proceedings of ATEC*, 2005, pp. 399–402.

[166] "Intel RDT Library [online]," Available: https://github.com/intel/intel-cmt-cat/tree/master/lib, accessed: 2019-08-26.

[167] "Kunpeng BoostKit for Virtualization [Online]," Huawei Technologies Co., Ltd., Tech. Rep. Issue 11, June 2021. [Online]. Available: https://support.huaweicloud.com/intl/en-us/twp-kunpengcpfs/kunpengcpfs-twp.pdf

[168] Canonical Ltd, "Ubuntu manpage: stress-ng [Online]," Available: https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2020, accessed: 2020-09-30.

[169] ESnet, NLANR, DAST, "iPerf tool for network bandwidth measurements [Online]," Available: https://iperf.fr/, 2020, accessed: 2020-09-30.

[170] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency," in *Proceedings of SoCC*, 2014, pp. 1–14.

[171] S. Seabold and J. Perktold, "statsmodels: Econometric and statistical modeling with python," in *Proceedings of SCIPY*, 2010, pp. 92–96.

[172] J. Wakefield, "Non-linear regression modelling and inference," in *Methods And Models In Statistics: In Honour of Professor John Nelder, FRS*. World Scientific, 2004, pp. 119–153.

[173] S. Dodd, A. Bassi, K. Bodger, and P. Williamson, "A comparison of multivariable regression models to analyse cost data," *Journal of Evaluation in Clinical Practice*, vol. 12, no. 1, pp. 76–86, 2006.

[174] M. Valbuena, D. Sarabia, and C. de Prada, "A Reduced-Order Approach of Distributed Parameter Models Using Proper Orthogonal Decomposition," in *Proceedings of ESCAPE*, 2011, pp. 26–30.

[175] E. López-González and M. Ruiz-Soler, "Análisis de datos con el Modelo Lineal Generalizado. Una aplicación con R," *Revista española de pedagogía*, vol. 69, no. 248, pp. 59–80, 2011.

[176] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11–25, 2015.