Concurrency and Computation: Practice and Experience

WILEY

Research Article

EMC+: an Opportunistic Elasticity Method for Improving System Throughput and CPU Utilization in Cloud Data Centers

Juan Carlos Saez | Carlos Bilbao | Manuel Prieto-Matias

Facultad de Informática, Universidad Complutense de Madrid, Calle Prof. José García Santesmases, 9, 28040, Madrid, Spain |

Correspondence: Juan Carlos Saez (jcsaezal@ucm.es) | Carlos Bilbao (cbilbao@ucm.es) | Manuel Prieto-Matias (mpmatias@ucm.es)

Received: <day> <Month>, <year> | **Revised:** <day> <Month>, <year> | **Accepted:** <day> <Month>, <year>

Funding: Spanish MCIU, Grant/Award Numbers: PID2021-126576NB-I00 and PID2024-158311NB-I00.

Keywords: multicore processors, operating system, elasticity, Linux kernel, malleability, Quality of Service, parallelism

Abstract

Multicore architectures have become the prevailing processor design for general-purpose computing systems, enabling high performance across a wide range of commercial platforms. Technological advances have made it possible to integrate hundreds of cores on a single package, providing unprecedented computational capacity. In cloud data centers, efficiently exploiting this growing core count becomes crucial for improving resource efficiency, lowering operational costs, and boosting revenue. To achieve this, servers usually run multiple colocated cloud services alongside diverse workloads. However, fully utilizing all available cores while enforcing quality-of-service (QoS) constraints for critical services remains a major challenge.

This paper introduces a novel OS-level approach designed to maximize CPU usage in multicore servers through the opportunistic acceleration of elastic HPC workloads. These workloads are capable of dynamically adjusting the number of active worker threads/processes at runtime. Our proposal, referred to as EMC+, is an OS-driven elasticity manager for container-based environments that continuously estimates idle core cycles left by regular (inelastic) applications, and reallocates idle cores to elastic ones, even during short time intervals. The new EMC+ proposal builds on our previous work and broads the scope of OS-driven elasticity through several key contributions: the design of a new kernel-level container-management framework, efficient operation on large-scale multicore platforms, the integration of QoS concerns into elasticity exploitation, and a comprehensive experimental analysis using diverse workloads that combine cloud services and HPC applications based on different parallel programming models. Experimental results show that EMC+ speeds up elastic workloads by up to 2.3x (1.5x on average), while having minimal impact on the performance and QoS of colocated inelastic applications. Compared to the unmodified Linux kernel, which does not exploit opportunistic elasticity, EMC+ boosts average system throughput by 19%, while degrading inelastic application performance by only 2.2% on average, and increasing the occurrence of observation intervals with QoS violations by 3.1% on average.

1 | Introduction

The chip multicore processor (CMP) architecture stands today as the predominant design choice for general-purpose systems across multiple sectors of commercial technology. Specifically, CMPs constitute the *de facto* architecture for servers in cloud data centers[1,2,3,4], and are also essential building blocks of HPC platforms[5,6]. Thanks to striking advances in technology and processor design, CMPs already reach hundreds of cores. Take, for instance, the recent AMD EPYC 9965 processor, which features 192 cores[7].

Maximizing resource utilization, and particularly, making the most out of the available CPU cores, is paramount for increasing revenue and reducing utility costs in cloud data centers[8]. Accomplishing this is a big challenge, given that typical cloud workloads

– most of them latency-critical (LC), such as search engines or AI-based services – usually exhibit highly variable loads due to irregular diurnal patterns[4,3,9]. The colocation of multiple applications on the same physical server (aka multi-tenancy) is now a popular practice for minimizing resource idleness[3,10]. Colocation increases resource efficiency, but makes it more challenging to enforce quality-of-service (QoS) constraints of cloud services due to contention[11,10,12,9]. Despite this, cloud providers often launch in-house workloads on the same set of servers devoted to running customers' workloads[4,11] so as to improve resource efficiency further during periods of low server utilization.

In this work, we explore a complementary approach to maximizing resource usage that leverages the execution of *elastic* HPC and scientific workloads. Today, *elasticity* is one of the cloud's main properties[12] and is defined as *"the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"*[13]. Elasticity can be exploited *horizontally* – by increasing the number of application instances using multiple containers/VMs – or *vertically* – by dynamically increasing CPU and/or memory resources of existing instances[8]. Current cloud elasticity mechanisms are typically engaged upon substantial fluctuations in the application's load, based, for example, on the number of external requests received[8]. Contemporary elasticity mechanisms[14,15,16,17] provide the necessary APIs for expanding or shrinking the resources available to the application across multiple VMs or container instances, which may be mapped to the same node or to different nodes. Notably, this expansion procedure is engaged at coarse granularities[18,19,20], as it usually requires starting new VMs/containers or spawning new worker processes across one or multiple nodes[17,21].

This work leverages a fine-grained form of vertical elasticity that operates at the server level. Instead of dynamically adjusting allotted resources based on explicit application demands (like most cloud elasticity methods do[14,15,16,8,22,23]), we strive to effectively utilize cores that may go temporarily idle in a shared server, even for very short time periods. We particularly exploit the fact that cloud workloads often leave idle cores even under a high load. However, these idle cycles sometimes last only tens or hundreds of milliseconds, making it challenging to exploit them without impacting application performance and QoS.

To take on this challenge, we designed EMC+ –an OS-level elasticity manager– and implemented it in the Linux kernel. EMC+ exploits the parallelism inherently available in many HPC and scientific workloads[24] to dynamically adjust their number of active worker processes/threads based on the fluctuating idle core cycles on the system. This successfully boosts the performance of HPC applications, while maximizing CPU usage and increasing system throughput. Notably, our approach does so with minimal impact on the performance and QoS of the remaining applications running on the shared server system. To accomplish these goals, we leverage explicit interaction between the OS and elastic applications, and exploit a novel OS-level method for accurate and efficient detection of idle core cycles. While this work demonstrates the opportunistic acceleration of elastic HPC applications, any parallel CPU-intensive workload could potentially benefit from our approach by leveraging the proposed interaction mechanism for elasticity.

To enjoy extra CPU resources opportunistically with EMC+, applications must support a basic form of *malleability*: the capacity to utilize a varying number of workers/cores at runtime. In HPC and scientific programs, exploiting malleability tends to be more manageable than in other domains, due to the inherent application parallelism, coupled with the use of runtime systems, whose functionality could be augmented to facilitate malleability[21,17,25]. At best, malleability can be delivered transparently to unmodified applications via runtime-system support[26,25].

This article is a substantial extension of our earlier work[25], augmenting it with the following contributions:

1. Our earlier elasticity manager (EM)[25], only supported single-process applications running right on top of the OS, impeding the execution of typical cloud workloads. Our new proposal leverages a new kernel-level framework for container management (described in Sec. 6) that enables the execution of a wide range of cloud and HPC workloads, many of which leverage multiple processes and threads.
2. Via exhaustive experimentation with realistic cloud workloads, we identified key overheads and QoS-related issues of our prior elasticity mechanism[25]. To address these issues, we designed two new container-enabled versions of the elasticity manager: an eager throughput-oriented implementation (EMC), and a conservative QoS-friendly variant (EMC+). Both strategies are described in Sec. 6.1.
3. Our OS elasticity framework was profoundly redesigned to deliver efficiency and scalability on systems equipped with numerous cores. To this end we experimented with an AMD-based server platform – described in Sec. 5.1 –, which differs significantly from the Intel system used earlier[25], especially regarding memory hierarchy and core count.
4. Our previous work considered workloads consisting of inelastic long-running HPC/scientific applications and elastic OpenMP programs only. We now experiment with a wider spectrum of applications (see Sec. 5.2), including both containerized cloud-like and elastic HPC workloads (both OpenMP-based and non-OpenMP-based). Particularly, we tested with web and database servers, in-memory-caching applications, as well as AI workloads. Our experiments also consider a more ample set of elastic HPC programs: applications built on top of parallel frameworks (MPI and Python based) that required modification to exploit elasticity, as well as unmodified OpenMP applications that leverage elasticity transparently via our previously proposed malleability-enabled runtime system[25]. Sec. 6.2 showcases key barriers we had to overcome to exploit opportunistic elasticity beyond OpenMP applications.
5. We conduct an extensive experimental evaluation of EMC and EMC+ using an AMD EPYC 7742 processor. Results in Sec. 7 show that EMC+ accelerates elastic HPC workloads by a factor of up to 2.3x, improving system throughput by up to 47% w.r.t. the default Linux scheduler, while minimally impacting the performance and QoS of inelastic applications. These improvements come from EMC+'s opportunistic and smooth utilization of idle core cycles. EMC+'s dynamic core redistribution method also overcomes key limitations of the prior redistribution strategy[25] (adopted in EMC) which is unsuitable for QoS enforcement.

The remainder of this paper is organized as follows. Sec. 2 discusses related work. Section 3 briefly motivates the creation of

our proposal. Sec. 4 provides a high-level description of our earlier Elasticity Manager (EM)[25]. Sec. 5 introduces our experimental platform and the set of elastic and inelastic applications used. Sec. 6 covers the design and implementation of EMC and EMC+, and reports on our experience augmenting existing inelastic applications with malleability support. Sec. 7 covers the experimental evaluation. Lastly, Sec. 8 concludes the paper.

2 | Related work

Forms of elasticity. The development of parallel architectures in the last decade has led to the creation of dynamic computing environments that exhibit dramatic fluctuations in resource utilization at runtime[22,27]. This has sparked increasing interest in the development of adaptive applications, capable of efficiently leveraging a varying amount of system resources over time. In the HPC community, substantial efforts have been devoted to the development of *malleable applications*, which are crucial to reduce both resource idling and energy consumption[27,21,28,5]. In the cloud, adaptive workloads are most commonly referred to as *elastic applications*[8,22].

To properly frame the nature of our proposed elasticity method, we must underscore that it exploits a form of fine-grained, opportunistic, reactive, intra-node vertical elasticity: when the OS observes idle core cycles left unused by the currently running applications, it allows elastic applications to temporarily utilize those otherwise idle resources by running worker threads or processes activated on demand. Our approach is reactive rather than proactive, because the decision to wake up workers is based on observed resource availability rather than on workload prediction. This stands in contrast with proactive elasticity approaches that forecast future workload demand and provision resources in advance[29,30,16,31]. Our approach is also vertical rather than horizontal, because the system does not create additional application replicas, processes, VMs or containers; instead, it dynamically adjusts the amount of thread-level parallelism made available to an already running application.

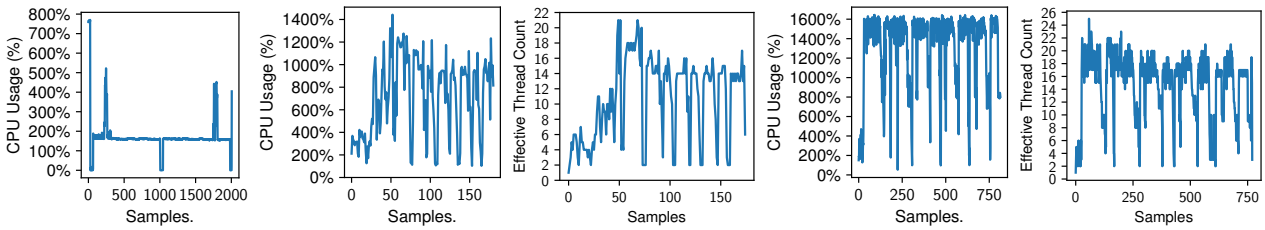
We should also highlight that our EMC+ proposal targets containerized applications, but employs worker threads or processes as the unit of elasticity. This approach substantially differs from other container-based elasticity (auto-scaling) approaches, where the unit of scaling is typically a container, or a microservice instance[29,30,16,31]. Moreover, our EMC+ proposal does not rely on migrating data, VMs or containers between nodes. Unlike migration-based elasticity, which moves execution state across computing nodes[32,33], our proposal strives to schedule additional active workers on idle cores in the same system, without moving application state or relocating execution across hosts.

Malleability in HPC. The state of the art in malleability exploitation in HPC is comprehensively reviewed by Tarraf et al.[21], who summarize the benefits that malleability brings to HPC, and identify key barriers to its practical adoption. These challenges include programmability, portability across elasticity-enabled resource managers, leveraging accelerators, and efficiently dealing with data redistribution upon dynamic worker adjustments – a relevant issue in large-scale applications[27].

Previous work on malleability in HPC takes a fundamentally different approach to ours for various reasons. Firstly, many works focus on easing the development of malleable applications with different programming models[28,34,26,25,5]. Instead, we explore the opportunistic OS-driven exploitation of idle core cycles on the system, allowing applications to benefit from it via a simple OS interface, which may be leveraged by the runtime system for transparent elasticity. Secondly, most HPC malleability approaches are not designed for cloud multi-tenant environments[27,5,35,25,21,36], so they do not deal with QoS constraints of latency-critical services, which EMC+ does consider. Lastly, the fine-grained nature and single-machine scope of our proposal stand in contrast with the higher granularity of other strategies, which operate at the cluster level or broader scales[37,28,38]. The different granularity is also evident in that these efforts may trigger elasticity operations with a potentially higher cost, such as spawning new workers, migrating workers across nodes, or performing container/VM orchestration at the data-center level. This makes these approaches largely orthogonal to ours.

Elasticity and resource management in the cloud. Prior efforts have explored solutions to align system capacity with user demand through elasticity in the cloud[8]. Most of these solutions operate in user space and focus on coarse-grained elasticity, typically making horizontal scaling decisions for groups of VMs[8,23] or containers[30,31,19,18]. These strategies are driven by explicit, demand-based application requests, and are primarily designed to support client-server applications by dynamically adjusting the number of VM/container instances in response to client load. Instead of reacting to long-term client demand fluctuations (on the order of minutes)[39,19,18], we adopt an opportunistic strategy: we capitalize on the inherent parallelism of HPC/scientific applications to make use of momentarily underutilized resources, even for a few hundred milliseconds. Thus, our OS approach addresses a different use case of elasticity than these demand-driven solutions. Notably, cloud providers also offer specific user services to launch low-priority batch workloads opportunistically, taking advantage of unused capacity at significant cost savings[40,41]. However, unlike EMC+, these services do not enable the exploitation of opportunistic vertical elasticity in applications; instead they employ rigid VM instances (i.e. with a fixed amount of resources) that can be stopped, and reallocated at any time to a different node by the cloud provider.

Some recent works have proposed diverse solutions to deliver QoS enforcement and improve resource utilization under colocation in cloud data centers[4,3,11]. However, they all follow a substantially different approach to ours, especially regarding two main aspects. Firstly, none of these strategies exploit application elasticity to improve system throughput. Instead, they run a variable number of inelastic best-effort (BE) jobs (which can be stalled if necessary) together with QoS-constrained workloads. This enables cloud providers to improve resource efficiency during periods of low server utilization by launching some of their own production workloads, many of which fall under the BE category[4,42]. Secondly, the resource management approaches proposed in[4,3,11] all require receiving continuous feedback on application-level metrics from QoS-constrained applications – such as current tail latency – to function. This requirement makes these policies unsuitable for public clouds, where applications are treated by the resource manager as black boxes that do not expose any kind of application-level feedback in real time[9,2,1]. By contrast, our EMC+ proposal follows a *black-box* approach amenable to public clouds: it operates without any application-level feedback from QoS-constrained applications.



(a) memcached CPU usage (b) IMA's CPU usage (c) IMA's ETC (d) GA's CPU usage (e) GA's ETC

FIGURE 1 | CPU usage and Effective Thread Count (ETC) observed over time for different applications on our platform.

Other related techniques. OS-level solutions for elasticity exploitation are scarce[25,12]. A related contribution in this area is the work by Huang et al.[12], which proposes various OS optimizations for efficient application execution under thread oversubscription. As opposed to our work, Huang's proposal does not dynamically adapt the number of active workers in elastic programs or exploit OS-application cooperation for improved throughput via opportunistic elasticity under QoS constraints.

Lastly, we should highlight that, while our work exploits elasticity as a means to maximize CPU utilization, other recent research has demonstrated that malleability can be also leveraged to increase resource usage and reduce contention in other shared resources, such as GPUs[43] or the file system[36]. These proposals are largely orthogonal to ours. Particularly, Sanchez-Checa et al.[36] present a malleability strategy that operates at the cluster level, and relies on application-specific performance models to drive file-system related elasticity decisions; this stands in contrast with our fine-grained OS-level intra-node approach designed to deal with general-purpose workloads, consisting of applications not seen beforehand. Villarrubia et al.[43] leverage NVIDIA's Multi-Instance GPU (MIG) technology to adjust the amount of resources assigned to a task before scheduling its execution on a GPU, potentially shared with other co-running tasks. Crucially, the proposed GPU-oriented scheduling strategy relies on offline profiling (applications are known by the scheduler) and does not adjust the amount of resources as a task runs (unlike what EMC+ does), as this type of dynamic reconfiguration is not currently supported by MIG.

3 | Motivation

Our proposal exploits the fact that typical cloud workloads may leave idle core cycles, even when subject to high client demands. To illustrate this observation, Fig. 1(a) shows the CPU utilization over time of `memcached` – a high-performance, distributed memory caching system[44] – running alone on our AMD-based platform, described in Sec. 5.1. In this experiment, we ran 4 `memcached` server instances with 4 workers each (16 threads in total) in a 16-core Docker container. CPU usage was gathered with the `systemd-cgtop` tool every 150 ms. In our setting, we issue 150K client requests per second (RPS), a value that stresses the allocated container resources, while ensuring that the service's tail latency does not exceed 1ms – a typical QoS constraint in `memcached`'s benchmark[44]. QoS violations begin to become apparent on our platform with slightly higher RPS values. Fig. 1(a) reveals that CPU utilization is below 200% most of the time, which is very far from the maximum attainable usage in a 16-core container (1600%). Low CPU-utilization scenarios like this could be exploited to improve system throughput by leveraging the available idle core cycles to run threads/vCPUs of other applications/VMs. However, as we demonstrate in Sec. 7.4, aggressively stealing idle cores leads to frequent QoS violations, thus making it challenging to effectively utilize idle cycles while enforcing QoS. Our EMC+ proposal addresses this issue.

We observed that relying solely on the CPU utilization metric leads to underestimating the number of idle cores that an application leaves, so it constitutes a misleading metric for CPU-usage optimization decisions. Take, for instance, the CPU usage over time (Fig. 1(b)) of `in-memory-analytics (IMA)` – an Apache Spark benchmark from Cloudsuite[45]. In the experiment, we ran `IMA` inside a 16-core container, where no user-imposed thread-to-core bindings were used, thus letting the Linux scheduler decide where to run each thread. As shown in Fig. 1(b), CPU utilization is below 1200% most of the time, still far away from the maximum attainable value here (1600%). This suggests that the application typically leaves a fraction of idle CPU cycles equivalent to 4 cores (i.e., the remaining 400% to reach maximum utilization).

Fig. 1(c) shows `IMA`'s *Effective Thread Count* (ETC) over time, as measured by our OS-level elasticity manager. The ETC – formally defined in Sec. 4.2 – is a conservative metric, which we determine by tracking the number of runnable (non-blocked) threads in an application/container during a given monitoring interval. In this scenario – with no user-imposed thread-to-core mappings – the metric allows us to approximate the average number of cores that the application uses in practice. The ETC metric for `IMA` (Fig. 1(c)) reveals that it spends a substantial amount of time with at least 14 active threads; because the kernel balances the load automatically, this means that 14 cores are used simultaneously most of the time. Notably, the application also goes through a few execution phases with oversubscription (ETC $>$ 16). These phases go unnoticed when using the CPU utilization metric (Fig. 1(b)), which does not even reflect the maximum attainable value (1600%) during the same monitoring interval. So, in general Linux's CPU utilization metric alone makes it difficult to accurately determine the number of idle cores that an application leaves, and it tends to underpredict it. Figs. 1(d) and 1(e) show the same trend with the CPU utilization and ETC metrics, respectively, for Cloudsuite's `graph-analytics` (GA) benchmark, described in Section 5.2.

We should highlight that the ETC metric cannot be tracked from user space on Linux, as the kernel does not expose to user space the necessary fine-grained monitoring information to collect it. In general, this requires awareness of thread transitions between runnable and non-runnable states, which are only visible within the OS kernel[46,25], where our elasticity manager operates.

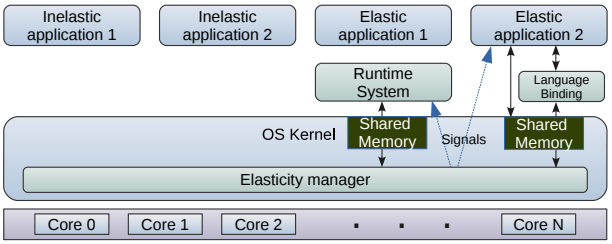


FIGURE 2 | System overview

4 | Overview of the elasticity framework

This section begins by providing a high-level view of our elasticity framework, placing a special emphasis on the inner workings of EM, our prior elasticity strategy[25]. Then, it describes the mechanism used by EM to determine the number cores left unused by an application. Lastly, it presents the communication interface exposed by the OS kernel to applications to leverage elasticity.

4.1 | Framework outline

Fig. 2 depicts the intended use scenario of our proposed framework and the interactions between applications and the OS kernel. The workload consists of multiple applications, running either directly on top of the OS or inside a separate container; the newly introduced support for containers is described in Sec. 6.1. Our framework includes an OS-level Elasticity Manager (EM) that distributes cores across applications and triggers elasticity actions upon fluctuations in the amount of utilized CPU resources. As shown in Fig. 2, some of the applications are *elastic*. They are initially assigned a set of dedicated cores, but can opportunistically use additional ones when available. To this end, they must interact with the kernel-level EM. To leverage opportunistic elasticity, the application must be able to adjust its number of active worker processes/threads at runtime based on the current CPU-resource allocation. This kind of elasticity support could be incorporated into existing programs by changing their source code – possibly requiring language-specific bindings for low-level interaction with the OS – or by using a custom user-space runtime system that brings malleability automatically to unmodified software. The remaining applications – referred to as *inelastic* – are conventional, potentially legacy, programs that do not interact with the OS to leverage elasticity. These unmodified applications, whether single-threaded or multithreaded, run on a fixed set of cores established by the user or by the cloud's resource manager. When underutilized, the EM may temporarily yield some of these cores to elastic applications.

The EM operates on top of Linux's scheduler, by monitoring the typical number of unused cores, and dynamically assigning available cores to elastic applications. While monitoring idle cycles is done continuously by the EM, core allocation decisions are made periodically, at a configurable rate, as discussed in Sec. 4.2.

To illustrate how the EM dynamically distributes idle cores among elastic applications, let us consider the simple step-by-step example shown in Fig. 3. In this scenario, an elastic application and an inelastic one run on a hypothetical 8-core system, with four cores initially assigned to each program (as depicted in Step 1). In the example, the number of active threads from the inelastic application (in red) varies over time, hence leaving idle cores sometimes. Elastic applications initially spawn as many workers

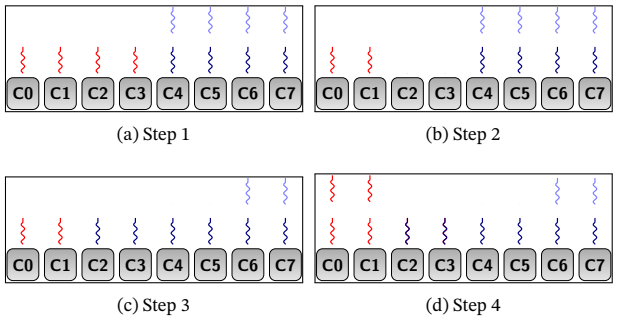


FIGURE 3 | Two applications running on a hypothetical system with 8 cores (C0–C7) under EM. Threads from the inelastic application are shown in red, whereas blue threads belong to the elastic program. Inactive workers are shown in light blue.

as the total core count, and dynamically wake up (or block) the necessary workers to utilize the cores allotted by the EM. This makes it possible to quickly populate idle cores that become available just for short time periods. For instance, in Step 1, just half of the workers of the elastic application (in blue) are active.

When an inelastic application consistently leaves cores idle, the EM yields extra cores to elastic programs so as to maximize CPU utilization. Steps 2 and 3 in the example reflect this fact, where the two idle cores left by the inelastic application are temporarily assigned to the elastic program, which populates them with two newly unblocked workers. To make this happen, three key actions are cooperatively performed by the EM and the application. First, because a change in the elastic application's core allotment is in order, the EM tells it how many cores it may use. To this end, we employ a per-application memory region shared with the OS kernel, together with an asynchronous notification (signal) that the kernel delivers to the application. Sec. 4.3 elaborates on this communication mechanism. Second, the EM alters the affinity mask of all threads in the elastic program, causing them to naturally expand over all assigned cores (as a result of the built-in functionality of the Linux load balancer). In a similar vein, EM also adjusts affinity masks in the inelastic application, so that it is packed in fewer cores based on its average runnable thread count. Third, the elastic application or malleability-enabled runtime system unblocks the necessary workers to match the allotted CPU resources.

When an increase in the CPU utilization of inelastic applications is observed, the EM reclaims cores that were temporarily assigned to elastic programs. This is depicted in a hypothetical transition from Step 4 back to Step 1 in Fig. 3. To accomplish this while minimizing the impact on inelastic-application performance, the EM packs all threads in elastic applications onto the new set of allowed cores, by altering thread's affinity masks. In addition, it properly signals elastic applications and updates the aforementioned shared-memory region to reflect the new core allocation. Note that the application or malleability-enabled runtime system is ultimately responsible for blocking some active workers to match the currently assigned CPU resources.

The example above abstracts much of the complexity that must be handled in a realistic setting. As we show in Sec. 4.2, determining the typical number of used cores is not as simple as monitoring CPU usage on individual cores. Moreover, typical cloud workloads may spawn hundreds of threads, which may block and wake

up frequently. This substantially complicates dynamic core allocation due to the associated overheads. Sec. 6.1 delves into this and other challenges addressed by our new proposal.

4.2 | Monitoring the idle core count

Accurately determining the number of idle cores left by inelastic applications is a critical aspect of EM. To do this, it continuously monitors thread transitions between runnable and non-runnable states (e.g., when a thread goes to sleep due to I/O, terminates, or wakes up from a blocking operation). Crucially, in typical cloud workloads, like those we experimented with (see Sec. 5.2), threads block when not doing useful work, thus making it easier to measure idle core cycles from the OS.

In reacting to thread transitions between runnable and non-runnable states, the EM tracks the time that each application spends with different runnable thread counts. This information is stored in a per-application *activity vector*, where each i-th entry stores the time (T_i) that the application spent with i runnable threads. An application's activity vector is updated when any of its threads becomes runnable or non-runnable; our implementation captures these OS-kernel events via a set of callbacks provided by the PMCSched framework[47]. When the system is idle, these callbacks are not called. Notably, all activity vector entries are reset during EM's periodic activation, which takes place each `em_period` ms, a configurable parameter. Resetting all entries is efficiently done by clearing an associated bit mask.

To determine an application's "typical" thread count – an approximation for the average number of cores it uses –, the EM employs the Effective Thread Count (ETC) metric. This metric is computed for each active application during EM's periodic activation. Specifically, the application's ETC is defined in terms of its activity vector as follows: $\frac{\sum_{i=1}^{M} T_i * i}{em\_period}$, where `em_period` is the elapsed time between two consecutive periodic activations, and M is the maximum thread count registered for the application during that observation time.[†]

During its periodic activation, EM approximates the number of cores left unused by each inelastic application. This is done by subtracting the application's ETC from the number of cores assigned to it, as established at the beginning of the execution. The estimated total idle cores are evenly distributed among elastic applications in the workload using EM's core redistribution algorithm. A detailed pseudo-code description of this algorithm can be found in our previous work[25].

4.3 | Kernel interface for elastic applications

To leverage opportunistic elasticity, applications must interact with the OS-level EM. As depicted in Fig. 2, this interaction is realized via shared memory and signals. The malleable application (or the malleability-enabled runtime system) must request the creation of the per-application shared memory region for effective communication between user and kernel spaces[‡]. The open-source PMCSched framework[47] –used to implement the EM in the Linux kernel–, provides support to create this kind of shared memory region; the application simply has to invoke the `mmap()` system call on a special file exported by PMCSched.

In our implementation, the shared memory region holds a simple two-field data structure. The first field, called `is_malleable`, is a flag that the application or runtime system sets to 1 to tell the OS that it supports malleability. The second field (`available_cores`) – exclusively modified by the EM– is used to inform the application of the maximum number of cores it may use. Every time this integer field is altered, the EM sends the SIGUSR1 signal to the corresponding application process. We opted to use that signal as it is a reserved one for user-defined purposes in the POSIX standard. The application or malleability-enabled runtime system must install the associated signal handler and carry out the necessary actions upon signal reception to adjust the active worker count. Despite the simplicity of the EM-userspace communication scheme, it could be easily augmented to conduct further optimizations based on user-provided hints. We elaborate on this aspect in Sec. 7.5.

As a proof of concept of the effectiveness of this EM-userspace communication scheme for elasticity exploitation, we use it in two different scenarios in this work: via transparent malleability support in the runtime system applied to unmodified programs, and via manual changes in existing applications.

The first scenario exploits the GNU OpenMP runtime system, which we augmented with malleability support, as described in detail in our previous work[25]. This implementation targets OpenMP applications leveraging `parallel for` constructs, and triggers malleability actions within runtime-system calls in charge of loop scheduling. Whenever a worker asks the runtime for new work to be completed (i.e., pending loop iterations), the worker may be assigned work and proceed – when active – or remain blocked in the runtime call – when inactive – until more active workers are needed. This approach allows fine-grained elasticity control and enables us to transparently deliver malleability to a wide range of unmodified OpenMP applications.

The exploration of the second scenario is one of the new contributions of this work on top of[25]. Sec. 6.2 sheds light on some of the challenges of manually augmenting existing inelastic applications with the proposed elasticity features.

5 | Experimental platform and applications

The key design features of our EMC+ proposal (described in Sec. 6) have emerged as a result of extensive experimentation with our earlier EM, but in an experimental setting that differs substantially from that of our previous work[25]. Specifically, we now run a richer set of workloads with a wider range of application types (including cloud-like applications and HPC/scientific programs) on a different multicore platform with a larger core count. Due to the enormous impact that this experimental setting had on the design of our proposal, this section anticipates the description of our experimental platform and the set of applications used.

[†] In our implementation, activity vectors have one entry per core in the socket, plus an extra entry to track periods when no threads are runnable. When the number of runnable threads exceeds the number of entries in T_i (i.e., under oversubscription in the socket), EM records the corresponding time in the last entry, which tracks full utilization time (no idle cores).

[‡] Allowing an application to communicate with the OS kernel via shared memory to exchange scheduling-relevant information is inspired by the `schedctl` feature of the Solaris operating system[47].

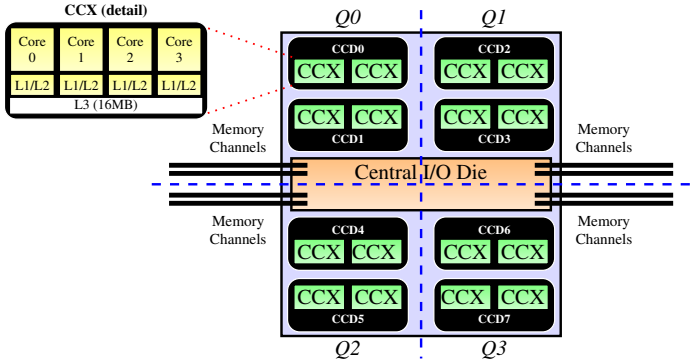


FIGURE 4 | Design of the 64-core AMD EPYC 7742 processor.

5.1 | Experimental platform

Our dual-socket experimental platform integrates two 64-core AMD EPYC 7742 processors, with a chiplet-based design. Cores in the same socket are organized into 16 four-core groups, each group featuring a separate L3 cache. The cache hierarchy of this AMD platform differs substantially from that of the system used in our earlier work[25], featuring a 16-core Intel processor, where all cores share the L3 cache. Nevertheless, the implementation of our new proposal also works without modification on processors where the cores within a socket share a common L3 cache.[§]

Fig. 4 depicts the organization of the AMD EPYC 7742 (Zen2) processor, consisting of 9 dies. The central I/O Die provides access to main memory and I/O. The remaining dies – named *Core Complex Dies* (CCDs) – include cores and caches[49]. The CCD's design differs across the different generations of the Zen microarchitecture. In our platform, each CCD is divided into two groups, each one featuring 4 cores with a 16 MB shared L3 cache. AMD refers to each of these core groups as a *Core CompleX* (CCX).

Our platform allows the system administrator to partition the various CCDs (cores and L3 caches) and memory channels in the socket into up to 4 "subnodes", represented by the 4 quadrants in Fig. 4 (labeled *Q*0 to *Q*3). Each quadrant includes 2 memory channels and 2 CCDs. Resource partitioning can be configured both *statically* via BIOS settings, and *dynamically*, via OS-provided memory-interleaving support. In our experiments we adopt a combination of both methods. A static partitioning mode can be enforced when the machine boots. Each supported mode exposes a different number of NUMA nodes per socket (NPS) to the OS[49]. Particularly, our platform supports 3 BIOS-managed modes: NPS1, NPS2 and NPS4. Under NPS1, all cores in the socket along with all memory banks connected to every channel make up a single NUMA domain; memory is automatically interleaved across the 8 memory channels. In NPS2, two NUMA domains are exposed to the OS, the first one consisting of resources in the Q0 and Q1 quadrants, and the second one encompassing Q2 and Q3; memory is interleaved across the 4 memory channels within each NUMA domain. Lastly, under NPS4, each quadrant is exposed as a NUMA domain to the OS, and memory is interleaved across both quadrant-specific memory channels.

[§] Recent Intel Xeon Scalable processors support Intel Sub-NUMA Clustering (SNC), including the older Intel Xeon Gold 5218 CPU used in our earlier work[25]. In that processor model, SNC exposes up to two NUMA/locality domains per socket by associating subsets of cores and LLC slices with different memory controllers/channels[48]. This mechanism is therefore conceptually similar to the NPS2 configuration of our AMD EPYC platform described later in this section.

Each NPS*i* configuration presents its own advantages and drawbacks[49]. NPS1 is usually the most suitable configuration for scalable HPC workloads, allowing a single application to leverage the socket's available memory bandwidth using all memory channels simultaneously. However, it is not the best choice regarding application isolation due to potential memory-bandwidth contention. Conversely, NPS4 is the configuration with the best isolation capabilities, as, by default, applications running on different quadrants access local memory pages solely through the quadrant-specific channels, hence not causing bandwidth contention in other quadrants. This configuration is more suitable for cloud workloads, where isolation is paramount for QoS enforcement[4,3]. In our experiments we use the NPS4 mode, but engage memory interleaving across the 4 quadrants – with `numactl` on Linux – just for elastic HPC workloads. The combination of these HW/SW settings allows us to leverage NPS4's isolation capabilities for inelastic applications, while enjoying increased memory bandwidth for elastic HPC workloads.

5.2 | Applications

5.2.1 | Inelastic Applications

In this work we employ seven cloud benchmarks:

Data-caching is an LC benchmark from the Cloudsuite 4.0 collection[44]. It uses `memcached`, a high-performance, distributed memory caching system that helps speed up critical cloud services by reducing database load. We had to modify the workload generator that comes with this benchmark to support over 4 simultaneous `memcached` servers.

Data-serving-relational is also part of Cloudsuite[45]. It is an LC benchmark that mimics the TPC-C workload using the PostgreSQL database management system.

Nginx is an LC benchmark that is part of the Open Benchmarking (OB) initiative[50]. It generates a configurable rate of client requests to the popular `nginx` web server. To allow the gathering of detailed and periodic latency statistics – not provided by the original benchmark – we employ a variant of the `wrk2` web workload generator created by Chen et al.[3].

Node.js is based on OB's Node.js Express load test[51], which performs requests to a REST API. For request generation, we use the aforesaid `wrk2` variant, instead of Node.js's `loadtest` tool, which does not report detailed latency statistics.

In-memory-analytics is a benchmark that builds a ranking of preferred movies based on a given user profile, by using a collaborative filtering algorithm that operates on a large in-memory dataset. This Cloudsuite benchmark[45] leverages Apache Spark's MLlib, a scalable machine learning library.

Graph-analytics is another Spark-based benchmark from Cloudsuite[45]. It performs graph analytics on a large Twitter dataset by using the Apache GraphX library, which enables graph-parallel computation. Notably, three variants of the graph-analytics benchmark exist, all differing in the underlying graph-processing algorithm used[45]: PageRank (`pr`), connected components (`cc`), and triangle count (`tc`).

DeepSpeech[52] converts a three-minute audio recording into text by employing a machine-learning model. The processing is done with Mozilla's DeepSpeech open-source speech recognition engine, based on TensorFlow.

A common feature of these benchmarks is that they all fail to achieve full CPU utilization even when (i) leveraging as many worker threads/processes as the number of assigned cores, and (ii) being stressed with a large number of client requests (this applies to LC applications only). As a result, workloads including these applications make perfect candidates for opportunistic CPU-usage optimization with our elasticity manager. To complement these applications, we also experimented with other inelastic programs with scalability bottlenecks used in [25]: `BLAST`, a bioinformatics application; `FFTW3D`, a scientific benchmark performing the FFT; and `swaptions`, a financial workload part of the PARSEC suite. These applications go through phases with a limited amount of thread-level parallelism due to the presence of load imbalance and/or inherently serial initialization stages.

5.2.2 | Elastic applications

Many of the elastic applications used in this work are unmodified OpenMP programs that leverage our support for transparent elasticity implemented in the runtime system [25]. Particularly, we used the following OpenMP applications: `EP` – from the NAS Parallel Benchmarks (NPB) –, `heartwall`, `particlefilter` and `sradv2` – from Rodinia, as well as `bonds` and `repo` – from the FinanceBench suite [53]. These benchmarks were chosen after conducting an extensive analysis on the scalability of a wide range of applications, similar to the one conducted in [25]. Clearly, non-scalable applications fail to benefit from extra cores allotted opportunistically. The aforementioned benchmarks exhibit good scalability on our AMD platform.

As a new contribution of this work, we built two additional benchmarks that serve as a proof of concept for elasticity exploitation with other parallel programming models beyond OpenMP. The challenges that arose in integrating our new proposal with these parallel programming models are described in Sec. 6.2; the description of these additional benchmarks is provided below:

PBBCache-elastic is an elastic benchmark we built on top of PBBCache [54], a Python-based open-source simulation and rapid prototyping tool of cache-partitioning policies. PBBCache is equipped with a parallel branch-and-bound optimization engine enabling to efficiently determine the cache-partitioning solution that optimizes different objective functions. This engine leverages a distributed master-slave strategy implemented with *ipyparallel* [55], where multiple worker processes explore different parts of the search space in parallel. In creating the elastic benchmark, we augmented PBBCache's functionality to use a dynamic number of active workers at runtime – as dictated by our elasticity manager – when searching for the optimal solution.

MPI-elastic is a synthetic master-slave C++ MPI benchmark built with OpenMPI, designed to evaluate elastic scaling in highly-parallel HPC workloads. In this benchmark, multiple worker processes execute independent matrix multiplication tasks dynamically submitted from the master. Large-scale independent matrix multiplications are a cornerstone of many real-world HPC applications, including neural network and LLM training, computational fluid dynamics or Monte Carlo-based financial modeling.

6 | New elasticity manager

This section begins by presenting the new features and optimizations introduced in the enhanced elasticity framework proposed in this work, which supports cloud workloads. Then, it reports on our experience in extending opportunistic elasticity beyond unmodified OpenMP applications, which showcased key barriers we had to overcome. Lastly, we summarize how our new container-based elasticity framework can be used in practice.

6.1 | The EMC and EMC+ elasticity managers

Our new elasticity framework was implemented as a scheduling extension in the Linux kernel v6.2.16. This extension is bundled as a plugin of the PMCSched framework [47], which can be dynamically loaded in unmodified (unpatched) kernels via a customizable kernel module.

The design of our new elasticity framework has been shaped by the insightful feedback we obtained via extensive experimentation with typical cloud workloads in our AMD-based server platform. These extensive experiments showed us that the EM [25] is unsuitable for these workloads for numerous reasons, including its inability to handle multi-process applications, the huge overheads it introduces under high thread/core counts, and its aggressive behavior in populating idle cores, which causes performance degradation and frequent QoS violations.

In what follows, we describe the new design features (❶ to ❺) in our implementation, which includes two new elasticity strategies. EMC leverages EM's original core redistribution algorithm [25], but adopts the new features ❶-❸ described below. EMC eagerly populates idle cores with threads from inelastic applications, substantially increasing system throughput, at the cost of potential performance/QoS degradation of inelastic applications. The EMC+ strategy incorporates all new design features (❶ to ❺), providing a smoother and QoS-friendly elasticity policy.

❶ **Support for containerized applications.** Many cloud and HPC applications –such as those based on Apache Spark, Hadoop, MPI, etc.– spawn multiple processes, so they are not supported by our earlier EM [25]. Our new implementation relies on application containers to enable the execution of these workloads. In particular, to evaluate EMC and EMC+ (see Sec. 7) we employed workloads consisting of multiple applications containerized with Docker. To perform OS-level resource management in a container-based environment like this, we had to substantially augment the functionality of the PMCSched framework [47]. This new container support allows developers to seamlessly manage all threads in the same container as a single group, by leveraging a data structure that exposes a Docker container as a set of threads belonging to the same Linux *cgroup*. This structure can be handled from a kernel module in much the same way as a multi-threaded application, and enables developers to keep track of the container's runnable threads and other relevant metrics in different parts of the system – within the same socket, inside each CCX, etc. The new container-related support is already available in the latest release of the open-source PMCSched framework [56].

❷ **Per-socket elasticity manager.** On NUMA systems, our new implementation runs multiple independent elasticity-manager instances, each operating on a separate socket. Each instance is in charge of optimizing the socket's CPU utilization, by dynamically allocating available cores to elastic applications/containers mapped to that socket. On our platform, this per-socket design results in a more scalable implementation compared to the system-wide approach of [25], which targeted a significantly smaller evaluation system. Note that the high core count per socket on our

(a) Initial scenario (each application running with 8 cores) (b) EM assigns 3 extra cores to elastic application (c) EMC/EMC+ assigns 3 extra cores to elastic application

FIGURE 5 | Differences in the core redistribution method under EM [25] and EMC/EMC+ when yielding three cores from an inelastic application.

Application	Thread activations and deactivations / sec. (TADPS)	Maximum Thread Count (MTC)	Avg. Effective Thread Count (ETC)	Avg. ETC vs. total threads (ETCT)	Avg. ETC vs. assigned cores (ETCC)	Simultaneously active threads vs. total (SATT)	Different active threads vs. total (DATT)
`EP` (OpenMP)	31.75	33	31.28	94.79%	0.98	96.97%	97.06%
`bonds` (OpenMP)	27.65	33	28.40	91.84%	0.89	93.78%	93.78%
`BLAST` (POSIX Threads)	11.40	33	3.67	66.39%	0.11	66.45%	66.47%
`MPI-elastic` (MPI)	2114.58	104	31.58	30.37%	0.99	31.96%	32.94%
`graph-analytics` (Apache Spark)	15102.79	287	21.17	9.86%	0.65	15.75%	39.04%
`in-memory-analytics` (Apache Spark)	69298.60	373	10.67	4.13%	0.33	12.97%	47.22%
`PBBCache-elastic` (ipyparallel)	36737.96	1491	28.49	2.88%	0.80	4.96%	17.87%
`DeepSpeech` (TensorFlow)	2736.02	66	1.93	2.93%	0.06	45.97%	57.87%
`data-caching` (Memcached)	243263.97	72	10.98	16.66%	0.34	27.11%	45.27%
`nginx`	9375.53	129	3.00	2.32%	0.09	5.64%	7.71%
`data-serve-relational` (PostgreSQL)	15753.54	42	3.19	10.61%	0.10	34.40%	83.63%

TABLE 1 | Statistics on total and runnable thread counts for various applications running on a 32-core container in our AMD experimental platform.

platform (64 cores) makes it possible to accommodate several containerized applications, allowing multiple container instances with similar or even higher core counts than typical container or VM instances in commercial cloud environments [10]. In practice, container/VM instances in the cloud rarely span multiple sockets [2], so using independent per-socket elasticity manager instances constitutes a reasonable choice.

❸ **Reducing thread migrations and migration candidates.** As explained in Sec. 4, core redistribution among applications occurs during EM's periodic activation. Figs. 5(a) and 5(b) illustrate how our previous EM proposal [25] performs dynamic core allocation in a workload consisting of one elastic and one inelastic application. Essentially, it ensures that threads in both applications use a non-overlapping set of cores. To this end, EM's implementation changes CPU affinity masks of all threads in both elastic and inelastic applications by leveraging the `set_cpus_allowed_ptr()` function from the Linux kernel's internal API. After EM alters thread affinity masks in this way, the Linux load balancer automatically evens out the load across cores by conducting the necessary thread migrations.

Implementing core redistribution in this way proved successful in performing dynamic core allocation in our previous work [25], where only OpenMP and POSIX-Threads-based HPC and scientific applications were considered. However, a preliminary evaluation of our earlier EM on the AMD experimental platform revealed substantial performance degradation when using other kinds of workloads, including cloud services and other real-world parallel applications leveraging different programming frameworks such as MPI, ipyparallel, Apache Spark, etc. As the evidence discussed next reveals, this performance degradation is caused by severe load imbalance scenarios induced by EM's periodic core redistribution, which alters thread affinity masks in these applications, which feature hundreds of threads.

Table 1 summarizes some insightful statistics on the number of active/created threads in a subset of the applications presented in Sec. 5.2. These statistics were obtained after collecting traces with detailed kernel-level statistics, gathered while each application ran alone on the system inside a 32-core container. For the first three applications listed in the table (also used in our previous work [25]) as many as 33 threads coexist on the system at a time, as shown in the Maximum Thread Count (MTC) column. Here, the MTC matches the number of cores in the container plus one; the extra thread belongs to the single-threaded parent (shell) process used to launch the application command. By contrast, for the remaining applications a larger number of threads coexist in the system at the same time. Take, for instance, the two Apache-Spark-based benchmarks in the table, whose MTC values are 287 and 373, respectively.

To demonstrate the negative impact of EM's core redistribution [25] on such applications, we focus on one of the multiple workloads that were subject to severe performance degradation: a workload combining `graph-analytics`, an Apache Spark-based cloud inelastic application, and EP, an OpenMP program made elastic thanks to our transparent malleability runtime-level support [25]. Table 2 shows the average completion times[¶] of both applications when running together under Stock-Linux (i.e., vanilla Linux, with no elasticity support), and under our earlier EM approach enhanced with container support, and using `em_period=120ms`, as in EM's original work [25]. The results reveal that EM accelerates the elastic application by 66.2% w.r.t. Stock-Linux, but slows down `graph-analytics` (the inelastic application) by a factor of 4.06. We found that this huge performance penalty comes from EM's core-redistribution method, which in this case adjusts affinity masks of a very large number of threads within short time periods. This often causes severe load imbalance scenarios on the system, which take a long time to be resolved by the Linux load balancer, negatively impacting application performance.

To gather evidence of this issue, we employed the low-overhead `runqlat` tool [57], which continuously measures kernel-level thread scheduling latencies (i.e., the time elapsed since a thread is enqueued in a runqueue until it is chosen to run on a core), and dumps latency histograms at a configurable period. Large scheduling latencies constitute an indirect indicator of load imbalance, which becomes especially severe when many threads

[¶] Sec. 7.3 describes in detail the procedure to run multi-program workloads like this.

Application	Stock-Linux	EM[25] with container support
`graph-analytics`	63.89 s	259.16 s
EP	16.36 s	9.28 s

TABLE 2 | Average completion time of applications in a sample workload.

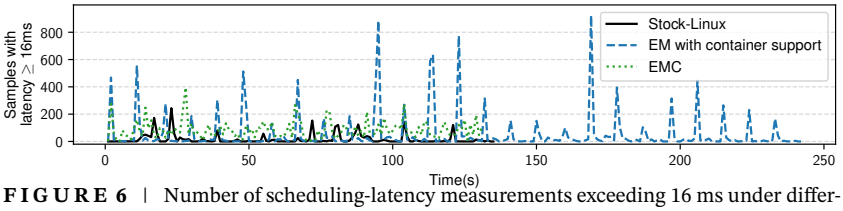


FIGURE 6 | Number of scheduling-latency measurements exceeding 16 ms under different scheduling strategies.

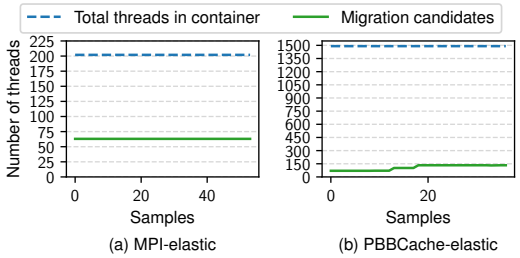


FIGURE 7 | Reduction of migration candidates in elastic benchmarks running inside a 64-core elastic container.

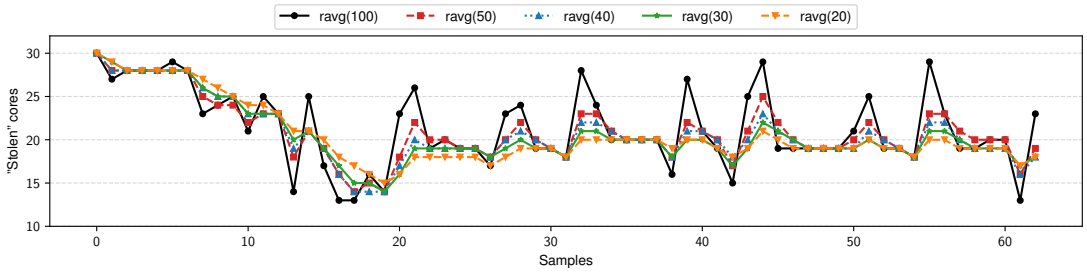
(samples) exhibit high latency values. Fig. 6 shows the number of thread latency samples that exceed 16 ms (four times the OS kernel's *tick* on our platform) every second during the execution of the aforementioned workload. Clearly, substantial spikes appear under EM, with abundant thread latency samples exceeding 16ms – over 800 samples at certain points. This indicates the existence of severe load imbalance. Conversely, for the execution under Stock-Linux –where two executions of `graph-analytics` complete within half the observation time– latency sample spikes are much smaller. Despite this, the creation and simultaneous activation of numerous threads during short-time intervals in `graph-analytics`, still introduce some high-latency spikes; these spikes reveal the existence of conflicting points in the execution where the Linux load balancer must rapidly address load imbalance situations.

To remove the severe load imbalance situations introduced by EM's core redistribution method, the EMC and EMC+ approaches include two optimizations. The first optimization is depicted in Fig. 5(c); unlike what EM does (Fig. 5(b)), EMC and EMC+ do not modify affinity masks of threads from inelastic applications upon core redistribution. Instead, they adjust affinity masks of threads from elastic applications only, which make inelastic and elastic applications share a subset of cores at certain times, as shown in Fig. 5(c). When threads from the elastic application are migrated to cores in the overlapping subset (after altering affinities), the Linux load balancer addresses the transient imbalance scenarios in this subset of cores only by moving threads from inelastic applications to idle cores. This approach removes the large performance penalty that EM incurs in this context. As Fig. 6 reveals, EMC introduce far fewer high-latency sample spikes than EM in the aforementioned workload. This reduces `graph-analytics`'s performance degradation to only 9% (average time is 68.9 s), while still speeding up `EP` by 32.8%.

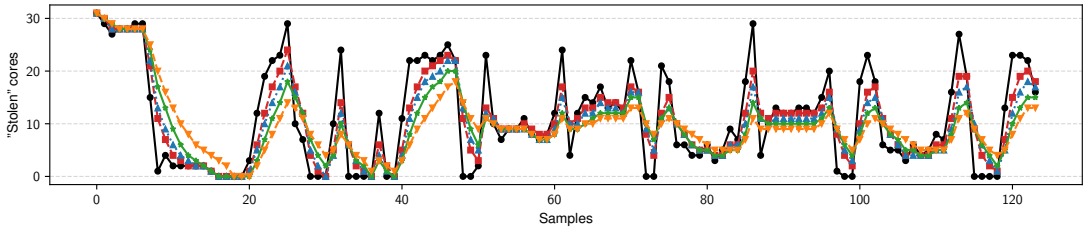
The second optimization was introduced to address other severe load imbalance scenarios that the first optimization could not resolve. These conflicting scenarios appeared in workloads including elastic applications built on top of MPI or Python-based parallel programming frameworks (such as ipyparallel[55]), which, unlike OpenMP programs, create many more threads than the number of cores in the container. As shown in the MTC column of Table 1, applications in this category –such as MPI-elastic and PBBCache-elastic– create many threads that coexist in the system at a time (nearly 1500 threads for the PBBCache-elastic). Crucially, many of these threads act as helper threads that remain blocked for a long time (e.g., those performing I/O, communication, garbage collection, profiling support, etc.). This is reflected in the table's DATT and SATT columns, which indicate that for these applications less than 33% of the threads created become runnable at some point during the observation interval, and the fraction of threads active at the same time (SATT) is much smaller than in OpenMP programs. Unfortunately, with those large thread counts, updating affinity masks for all threads in these elastic programs caused the same transient load imbalance scenarios as those described earlier. To overcome this shortcoming, the second optimization consists in altering CPU affinity masks just for a subset of threads in elastic containers: those threads that exhibit a CPU-intensive profile, and, hence, are likely serving as workers. Threads like these can be easily identified within the OS kernel by their large fraction of involuntary context switches. Fig. 7 shows the total threads in the application container vs. the actual thread migration candidates selected by EMC and EMC+ during core redistribution operations in two independent workloads. Clearly, our proposal substantially reduces the number of threads whose affinity mask is readjusted upon core redistribution, being less than 9% of the total thread count in `PBBCache-elastic`.

❹ **Smoothing aggressiveness of idle core redistribution.** As opposed to most of the scientific and HPC workloads we experimented with in[25], many real-world applications exhibit phases with very short CPU bursts, which lead to frequent thread activations and deactivations (i.e., transitions into and out the *runnable* state). To illustrate this aspect, Table 1 shows the number of thread activations and deactivations per second (TADPS) of several applications, where cloud-like workloads like `memcached` or `in-memory-analytics` exhibit an average TADPS in the order of tens of thousands of changes per second in the runnable thread count. These fluctuations also impact the ETC metric, causing very frequent and diverging core redistribution operations under EMC. As our experiments in Sec. 7.4 reveal, sudden changes in the ETC metric in a few applications make the EMC strategy aggressive, leading it to trigger frequent thread migrations. This degrades inelastic-application performance, but most importantly, causes frequent QoS violations because core redistribution operations take time to fully stabilize. Note that attempting to address this problem by enlarging the activation period (`em_period` parameter) is not effective, as our experiments in Sec. 7 reveal. This reduces the frequency of core redistribution operations, but also makes EMC less eager in reclaiming cores from elastic programs, which degrades performance.

To overcome this problem, our proposed EMC+ approach makes core-redistribution decisions based on a running average of the ETC metric maintained for each inelastic application, rather than employing the raw (most recent) ETC value. The per-application



(a) in-memory-analytics



(b) graph-analytics

FIGURE 8 | Cores "stolen" by EMC+ (*ncores - ravg*) with different values of the `ra_factor` parameter when running two applications on a 32-core container in separate runs. The numbers in parentheses indicate the `ra_factor` value used to calculate the running average (*ravg*)

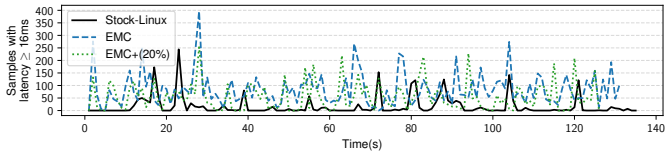


FIGURE 9 | Impact of the `ra_factor` parameter on the number of latency scheduling measurements exceeding 16 ms.

running average (*ravg*) is updated every activation period, by taking into consideration the raw ETC value (denoted as ETC_t) and the old running average value, calculated in the previous activation period: $ravg_t = ETC_t \cdot f + ravg_{t-1} \cdot (1 - f)$, where f denotes the contribution or weight of ETC_t in the calculation of the running average. In our implementation, f can be adjusted via the `ra_factor` configurable parameter, expressed as a percentage. The higher the percentage, the more aggressive the behavior of EMC+'s core redistribution algorithm.

To complement our extensive experiments in Sec. 7, which include numerous multi-application workloads under different choices of the `ra_factor` parameter, Fig. 8 illustrates how this parameter affects the degree of aggressiveness in core redistribution. Specifically, the figure shows how many cores reserved for two particular applications are temporarily reassigned ("stolen") by EMC+ to elastic programs for different `ra_factor` values ranging between 20% and 100%. When `ra_factor`=100%, the running average matches the ETC metric, so EMC+ operates like EMC under these circumstances. For this parameter choice, sudden changes in the ETC metric may cause large variations in core allocation between consecutive periodic activations, as spikes in Figs. 8(a) and Fig. 8(b) reveal. Conversely, using the running average with lower values of `ra_factor` allows us to effectively filter out high spikes, while avoiding dramatic changes in the number of re-assigned cores in consecutive periodic activations.

Overall, relying on the running average rather than the raw ETC provides a much smoother behavior of the elasticity manager, while still allowing it to reclaim cores quickly when necessary (i.e., by using a low `em_period` value). As our experiments in Sec. 7.4 reveal, EMC+'s ability to address spikes in the ETC metric is crucial for maintaining low scheduling latencies under elasticity exploitation, which is essential to ensure low performance degradation of inelastic applications. Fig. 9 shows the effect on the scheduling latency of using a low `ra_factor` value for the aforementioned two-application workload (`graph-analytics+EP`). As is evident, EMC+ consistently reduces the number of high-count latency spikes associated with core redistribution with respect to EMC, which relies on the raw ETC metric. Our extensive experiments in Sec. 7 demonstrate that this EMC+ feature is also crucial for QoS enforcement.

❺ **Checking representativeness of average runnable thread count.** An early evaluation of the ❶-❹ design features showcased another issue that became especially apparent when using one of the evaluated applications: `in-memory-analytics`. In this application, the average number of runnable threads observed during the activation period (i.e., the ETC metric) is not always representative of the typical number of cores an application uses. Particularly, the runnable thread distribution captured by the various activity vector entries (T_i) reflected that it sometimes spends more than 50% of `em_period` with a number of threads greater than the ETC (average). Fig. 10 shows the amount of time that `in-memory-analytics` spends with different numbers of runnable threads during one of the observation intervals, where the average runnable thread count is 9.85. However, in this case, the application spends over 55% of the monitoring interval with at least 11 runnable threads. Unfortunately, using the average here underestimates the fraction of CPU resources in use, which indirectly degrades the performance of this inelastic application[#].

We mitigate this problem by taking into consideration a more conservative prediction of the number of utilized cores, that is, a thread count for which the application spends a substantial time fraction F of `em_period`. Specifically, to determine this conservative prediction, we define the *CETC* metric: it is the highest runnable thread count k such that $\frac{\sum_{i=k}^{M} T_i * i}{\texttt{em\_period}} \geq F$, where F is a

[#] We conducted a similar analysis for the remaining applications considered, and found that this average-related issue appeared only rarely in just a few observation intervals. Hence, the optimization discussed here mostly impacts workloads including `in-memory-analytics`.

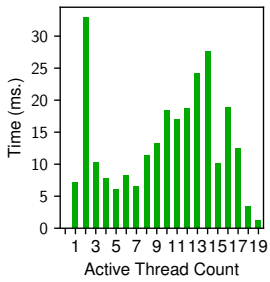


FIGURE 10 | Time spent by `in-memory-analytics` with different runnable thread counts during a 250 ms interval.

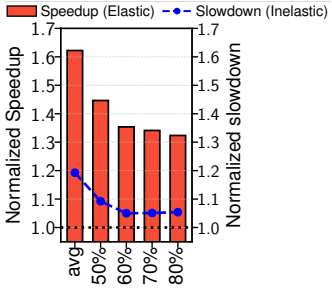


FIGURE 11 | Normalized speedup and slowdown for an application mix combining `in-memory-analytics` and `EP`

configurable fraction. If the ETC (average) exceeds *CETC*, EMC+ uses *CETC* for the calculation of *ravg* instead of the ETC.

Before carrying out our extensive experiments of Sec. 7, we conducted a sensitivity study to determine an appropriate value of this parameter. For brevity, we summarize the conclusions of this study by discussing the results for one of the evaluated workloads with a conservative configuration of EMC+ (with `ra_factor`=20%). In particular, Fig. 11 shows the results for a workload that combines the `in-memory-analytics` (inelastic) application and the `EP` (elastic) program; specifically, it reports the performance degradation of the inelastic application and the speedup of the elastic program w.r.t. Stock-Linux (no elasticity). In this experiment, we repeatedly launched the workload under different configurations: (i) with different values of the F factor (indicated as percentages in the figure), and (ii) with the optimization disabled (labeled as "avg"), namely, by using the ETC to determine the running average. Clearly, increasing the percentage allows us to trade some performance gains in the elastic program for less slowdown (as low as 1.05 when using the 60% to 80% settings). The full sensitivity study revealed that using $F = 0.6$ (60%) –our choice for the experiments of Sec. 7– provides a good trade-off between throughput and performance/QoS penalty. In general, the higher the value of F, the more conservative the behavior of EMC+ when yielding cores from inelastic applications to elastic ones.

6.2 | Elasticity exploitation beyond OpenMP

In our previous work [25] we only experimented with unmodified OpenMP programs made elastic via transparent runtime-system support. Here, we shed light on some of the challenges of manually augmenting existing non-OpenMP applications with the proposed elasticity features. To this end, we experimented with two additional programs: `PBBcache` –a parallel Python-based simulator – and `MPI-elastic` – a C++ MPI benchmark. Whereas Sec. 5.2 already described the type of processing these applications do, here we discuss how we made the applications elastic.

Both elastic programs exploit parallelism through multiple processes that communicate via message passing and follow a master-slave computation model. The master dynamically controls the number of active worker processes, aiming to match the worker count with the number of cores allotted by the elasticity manager. When making the changes needed to make the applications elastic, we had to overcome two main barriers: (1) ensuring that inactive workers remain blocked in the MPI program, and (2) devising a means to access the memory region shared with the kernel from Python code.

In the Python-based simulator, idle workers – managed by the *ipyparallel* distributed computing framework [55] – remain blocked, significantly simplifying the enforcement of the desired number of runnable processes. However, in the MPI program (implemented with OpenMPI), workers waiting for further work to complete busy-wait automatically instead of blocking. To address this issue, the master process has to explicitly block and wake up idle workers. In our prototype single-machine implementation the master accomplishes these actions efficiently by delivering the SIGSTOP and SIGCONT signals, respectively, to specific MPI worker processes.

The second barrier arose due to a limitation in Python's wrapper for the `mmap` system call – part of the `os` module –, which does not support invoking this operation on special files, such as the *procfs* entry our framework relies on [47]. To overcome this shortcoming, we developed a custom Python extension module in C that provides a high-level interface enabling seamless access to the aforementioned data structure shared with the kernel.

6.3 | How to use EMC+?

From a technical standpoint, running Docker-based containerized workloads with EMC+ is straightforward. As a prerequisite, the kernel module of the PMCSched framework (part of the PMCTrack [56] tool) must be loaded in the system, and the corresponding scheduling plugin that implements EMC+ must be activated. A PMCSched plugin can be activated by just writing to special files of Linux's *procfs* – details can be found in PMCSched's official documentation [58].

All interactions between EMC+ and user space occur via the `/proc/pmc/sched` special file, exported by PMCSched. For example, the value of each configurable parameter in our elasticity framework (e.g., `em_period`) can be safely set just by writing a "`<parameter_name>=<value>`" string to this file. The current values of all the configurable parameters can be retrieved by reading from that file. To ensure that EMC+ manages a particular application – whether elastic or inelastic –, the user or cloud-management software must first create the associated container (e.g., via Docker's CLI tools), and then activate the container's *cgroup* in the kernel, by writing the "`register_cgroup <PID>`" string to the special file, where PID is the process ID of the `init` process in the associated container. This PID can be easily retrieved via the `docker inspect` command.

Our EMC+ implementation automatically determines the set of cores that each application can use by examining the Linux `cgroup` structure of the corresponding container. All applications are initially treated as inelastic; as described in Sec. 4.3, each elastic application identifies itself as elastic afterwards by requesting the creation and proper initialization of the data structure stored in the memory region shared with the kernel.

7 | Experimental evaluation

To assess the effectiveness of EMC and EMC+, we employ containerized workloads that combine cloud-like and HPC applications running on the AMD server platform presented in detail in Sec. 5.1. All the applications used are described in Sec. 5.2.

Many of these applications are batch programs, whereas others are latency-critical (LC) services. For LC services we continuously measure client-request latencies, and the number of processed requests per second (RPS), whereas for batch applications we measure their completion time.

Our baseline for comparison is the vanilla Linux kernel v6.2.16 – referred to as *Stock-Linux* – which does not exploit elasticity. For the experiments with *Stock-Linux*, we do not load the kernel module that implements EMC and EMC+, to remove any potential interference. In our previous work [25] we already evaluated the impact of leveraging oversubscription on top of Stock-Linux, as a means to exploit idle cores without augmenting the OS functionality. As that work clearly revealed, oversubscription leads to severe performance degradation of inelastic applications. Moreover, it is known to make it very difficult to deliver QoS for LC workloads [10]. Therefore, we do not evaluate oversubscription scenarios in our experiments. Note also that none of the previous related works discussed in Sec. 2 exploit opportunistic elasticity when running cloud-like workloads, and relevant QoS-aware yet inelastic proposals [4,3,11] simply do not work in the black-box scenario EMC+ targets (i.e., inelastic applications do not provide any kind of real-time QoS feedback to the resource manager). Therefore, a direct experimental comparison with these proposals is not feasible in this context.

The remainder of this section is organized as follows. Sec. 7.1 presents the set of metrics used in our evaluation. Sec. 7.2 describes the workloads and evaluation scenarios considered. Sec. 7.3 presents the methodology to run the multi-application workloads, and the method to stress LC services with client requests. The discussion of our extensive experiments in the different evaluation scenarios can be found in Sec. 7.4. Lastly, Sec. 7.5 enumerates some limitations of our proposal that motivate potential enhancements planned for future work.

7.1 | Metrics

To quantify the performance degradation of each application a_i in an N-application workload $W = \{a_1, a_2, \cdots, a_N\}$, we use the *Slowdown* metric [59,6,54]. Specifically, let $Perf_{alone,a_i}$ be the performance of a_i observed when it runs alone on the system, and let $Perf_{res,a_i}$ be a_i's performance when running together with the other applications in W under a specific resource management policy. $Slowdown_{a_i}$ is defined as $Perf_{alone,a_i}/Perf_{res,a_i}$. To quantify the performance of an LC application, we use its RPS in the two scenarios (i.e., running alone and together with the other applications, respectively). By contrast, for batch applications we use the inverse of the completion time observed in the two aforementioned scenarios to measure performance.

To determine the system throughput when running a workload, we employ the widely-used STP metric [59,6,54,60], also referred to as *Weighted Speedup*. The STP can be expressed in terms of the slowdown of each application as follows:

$$STP = \sum_{i=1}^{N} \frac{Perf_{res,a_i}}{Perf_{alone,a_i}} = \sum_{i=1}^{N} \frac{1}{Slowdown_{a_i}} \quad (1)$$

7.2 | Workloads and evaluation scenarios

Fig. 12 shows the composition of the 70 randomly generated workloads used in our experiments, consisting of two or four containerized applications. Particularly, each 2-application mix (from M1 to M45) combines an inelastic application and an elastic program. Each 4-application workload (W1 to W25) is made up of three inelastic applications and one elastic program. To differentiate elastic from inelastic applications in the workloads, the name of elastic applications is denoted in *italics* in Fig. 12. The initial mapping of applications to different sets of cores in our platform is described in Sec. 7.3.

With these workloads we explore two evaluation scenarios. The first scenario comprises workloads consisting of 2 applications (from M1 to M45). The main goals of this first round of experiments are (i) to assess the sensitivity of EMC and EMC+ to the choice of the different configurable parameters, and (ii) to determine the best-performing parameter choices regarding throughput, elastic application acceleration and QoS enforcement. Our second evaluation scenario encompasses the 4-application workloads (W1 to W25), which we run under the best-performing EMC and EMC+ configurations – identified in the previous round of experiments. With this second set of experiments, we strive to analyze EMC+'s potential to accelerate elastic HPC programs, when opportunistically utilizing core cycles left unused by multiple containerized applications, while exploiting a potentially larger pool of idle cores.

7.3 | Experimental methodology

In conducting the experiments on our experimental platform – described in Sec. 5.1 – we devote one socket of the machine (CPUs 64–127) to run the clients that generate the requests to LC services; the other socket (CPUs 0–63) runs the multi-container workload, where elasticity is exploited. Notably, containerized applications and clients have their data and code placed in the memory banks associated with the socket where they run. In allotting disjoint sets of CPU and memory resources to containerized applications and clients, most interference is removed, thus allowing us to cleanly evaluate the raw potential of our proposal.

In running the workloads we follow a similar approach to that of [61,6], which focuses on keeping the system load constant throughout an experiment while factoring in the disparities in the duration of the various batch applications present in the mix. Specifically, we ensure that all applications in the workload are started simultaneously[‖]. When a batch program completes, it is restarted repeatedly until the longest batch application in the set completes at least three times and has run for at least ten minutes. When that happens, we stop the remaining applications, including the LC services. We then measure the STP by considering the RPS of each LC service throughout the experiment, as well as the geometric mean of completion times observed for every full execution of each batch program.

For the first evaluation scenario (2-application workloads), we used the initial container mapping shown in Fig. 13(a), where each application is allotted 32 cores, effectively utilizing two quadrants of the AMD EPYC 7742 processor (see Sec. 5.1). In contrast, for the second evaluation scenario (4-application workloads), the initial core assignment, shown in Fig. 13(b), allocates 16 cores per

[‖] Some LC services must run a warm-up phase to cache datasets in memory. Batch applications in the workload are launched when LC services' warm-up has completed, which is discarded for metric measurements.

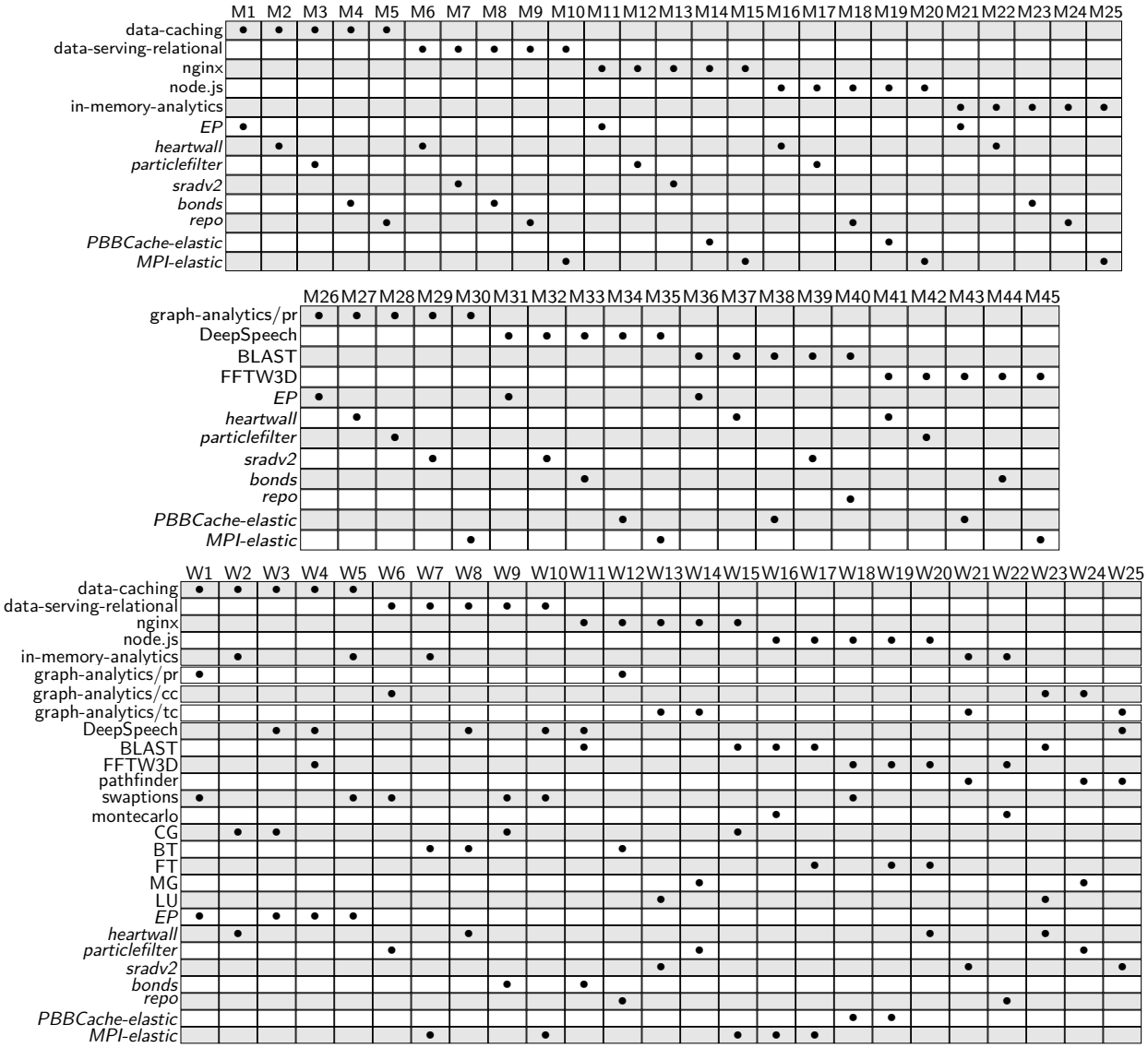


FIGURE 12 | Workload composition. A dot in a cell indicates that the application named in that row is present in the workload represented by that column. The names of elastic programs are highlighted in *italics* in the corresponding table labels.

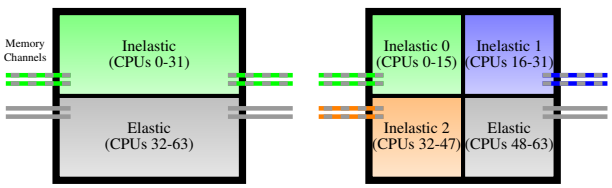


FIGURE 13 | Initial core distribution for (a) 2-application workloads, and (b) 4-application workloads.

container, corresponding to one processor quadrant each. Inelastic applications always run on their initially assigned cores, while the elastic program can opportunistically utilize idle cores from inelastic containers as dictated by the elasticity manager. Note that, in increasing the number of allotted cores and active workers in an elastic application, its demand for memory bandwidth may increase significantly, which could create a performance bottleneck[25]. To avoid exhausting the memory bandwidth through the memory channels associated with the initial quadrant(s) of the elastic program, we distribute its data across the four system quadrants by employing memory interleaving when launching the program. Thus, the elastic container can simultaneously leverage all the memory channels available in the socket, as depicted in Fig. 13. Conversely, data for inelastic programs is allocated in the memory banks of the corresponding quadrant, whose memory channels provide sufficient bandwidth for the explored applications. Although this static memory placement may cause some contention between elastic and inelastic applications, throughput gains outweigh the drawbacks. So, we opted to adopt this strategy when conducting our experiments. Exploring dynamic data placement techniques to mitigate contention effects under elasticity remains a promising direction for future research.

Table 3 shows the number of client requests issued per second (CRPS) and maximum latency we used for the latency-critical services included in the workloads, for the two core allocations. To determine these settings in our experimental system, we followed a methodology similar to that of Chen et al.[3]: by progressively stressing the LC service until it reaches a saturation point with the allocated system resources. Specifically, we ran the LC service alone on the system (using 16 or 32 cores) gradually increasing

TABLE 3 | Client requests issued per second (CRPS) and maximum latency for the containerized latency-critical applications.

	32 cores		16 cores	
Application	**CRPS**	**Max. Latency**	**CRPS**	**Max. Latency**
data-caching	300K	1ms	150K	1ms
data-serving-relat.	100	100ms	100	150ms
nginx	9.5K	10ms	7.4K	15ms
Node.js	480	10ms	400	10ms

the CRPS, while monitoring the latency distribution. Tail latency is measured at the 99th percentile for all services, except for the `data-serving-relational` benchmark, where the 95th percentile is used[45]. Maximum latency is defined as the tail latency at the critical inflection point in the CRPS-to-tail-latency curve, where a minor increase in CRPS causes a dramatic rise (several orders of magnitude) in tail latency. For multi-container workloads, we set the CRPS value just below that inflection point, which ensures that – when running in isolation – tail latency remains within the maximum threshold while approaching the lowest CRPS setting that would cause QoS violations (i.e., tail latency exceeding the maximum latency). Notably, although the various inelastic services use as many workers as the number of assigned cores, the CRPS setting does not always scale proportionally to the core count, due to other bottlenecks that become apparent in our setup. For example, `data-serving-relational` is primarily constrained by disk performance, with a single SSD used to store the underlying database regardless of the core allocation. Similarly, in the Node.js and Nginx benchmarks, higher CRPS values lead to client-side connection timeouts, limiting further CRPS increases. Lastly, note that, in calculating the slowdown metric for an LC service running in a multi-container workload, we factor in its actual RPS achieved, which could be smaller than the CRPS when the service cannot keep up with the target request rate.

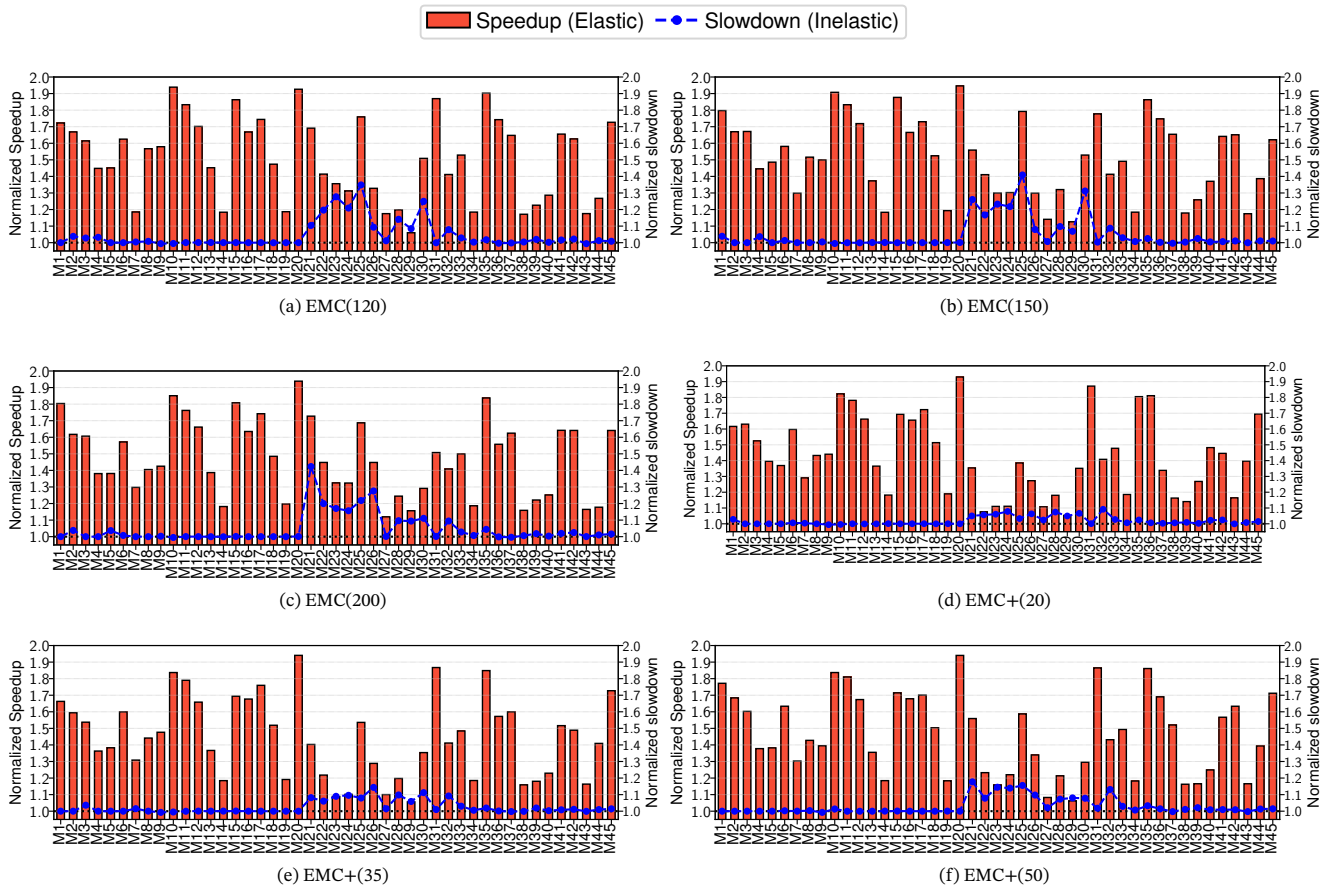
7.4 | Discussion

First evaluation scenario (workloads M1–M45). The results of the two-application workloads are shown in Figs. 14 to 17. These figures show just a subset of our extensive results, which include vast experimentation with multiple parameter values of EMC and EMC+. The showcased subset of results captures the most relevant across the different parameter choices.

EMC is sensitive to the choice of the `em_period` parameter, which establishes the activation period length (in ms). Figs. 14 to 17 show the results with three `em_period` values: 120, 150 and 200. The number in parentheses by "EMC" in the charts' legend and caption indicates the value of that parameter. Among the multiple `em_period` values we tested, we found that values below 120ms introduce excessive thread-migration-related overhead, while values above 200ms prevent EMC from reacting quickly when immediate core reclamation is required; in both extreme cases, the performance of some inelastic applications is severely degraded. EMC+'s performance is also sensitive to the choice of the `ra_factor` parameter, allowing it to control the degree of aggressiveness when stealing and reclaiming cores. The aforementioned figures show the results for EMC+ with 3 different `ra_factor` choices: 20%, 35% and 50%. The parenthesized number by "EMC+" denotes the `ra_factor` used. Note that for EMC+ we set `em_period` to 120ms, as it delivers the best performance. This choice allows us to keep the core stealing and reclamation rate under control thanks to EMC+'s smoothing mechanism, ensuring stability despite frequent periodic activations. Note also that the periodic core redistribution algorithm – executed on one core once every `em_period` – has very low overhead: in its most sophisticated form (EMC+) it takes 14.9μs on average to complete, and lasts no longer than 22μs in 85% of the algorithm invocations, which is negligible compared to `em_period`'s duration.

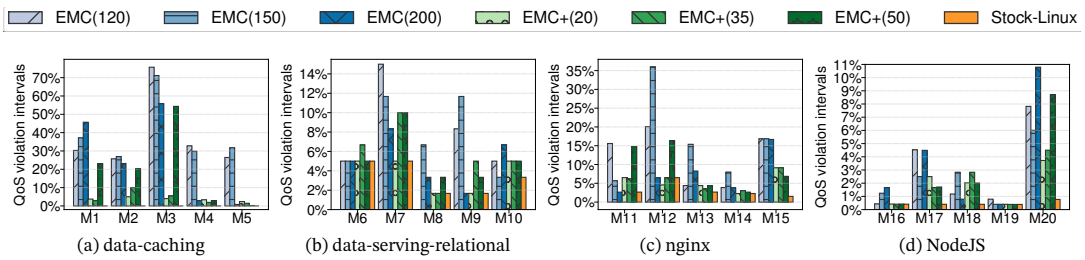
Fig. 14 shows the speedup of the elastic application and the slowdown of the inelastic workload w.r.t. Stock-Linux (no elasticity) for every M*i* pair. EMC yields much higher performance improvements for the elastic programs than EMC+ (up to 1.94× speedup vs. Linux), especially when using an `em_period` setting of 120ms or 150ms. For these two configurations, EMC achieves an average speedup of 1.5 vs. Linux (see Fig. 17(a)), just slightly higher than 1.43, the average value of EMC+(20) – the least aggressive strategy. Notably, for roughly 80% of the workloads, EMC obtains substantial speedups without significantly affecting the performance of the inelastic programs (slowdown below 1.05). However, accelerating the elastic program sometimes comes at the expense of some performance degradation of the inelastic application. Take for instance the M21-M25 or M30 workloads, where the inelastic application slows down by 1.25x or more under EMC. Conversely, thanks to its smoother behavior, EMC+(20) reduces this slowdown to 1.09 or less in these cases, in exchange for a more modest elastic-program acceleration.

The differences between EMC and EMC+ become even more pronounced when analyzing QoS aspects. Particularly, Fig. 15 shows the percentage of monitoring intervals in which QoS violations are observed, that is, when the LC service's tail latency exceeds its maximum latency. The QoS monitoring interval for `data-serving-relational (DSR)` is set to 5 seconds – the default value of that benchmark[45]; for the remaining services, tail latency (p99 latency) is measured every second. The latency results for the `data-caching` benchmark (Fig. 15(a)) are particularly revealing. EMC introduces QoS violations in up to 75% of the monitoring intervals – e.g., M3 under EMC(120). This is caused by EMC's aggressiveness when it comes to stealing and reclaiming cores upon sudden fluctuations in `data-caching`'s ETC, which lead to load imbalance due to frequent thread migrations. The Linux load balancer may take tens of milliseconds to resolve the transient oversubscription caused by these migrations, which causes worker threads of the `memcached` service to sit on the scheduler run queues for some time. This increases the client-request processing time beyond 1ms in some cases, causing QoS violations. While enlarging the `em_period` does not help mitigate this problem, EMC+ does address it effectively thanks to using the ETC's running average instead of the raw ETC to guide core stealing and reclamation operations. Clearly, the percentage of QoS violation intervals is reduced to 10% or less when under EMC+(20) and EMC+(35). While EMC+(50) does yield slightly higher average speedup and throughput (see Figs. 17(a) and 17(c)), using a 50% `ra_factor` becomes harmful when it comes to QoS enforcement. Figs. 15(c) and 15(d) reveal smaller percentages of QoS violation intervals for `nginx` and `node.js`, but a similar trend can still be observed: EMC+(35) and, especially, EMC+(20) consistently deliver fewer QoS violations. The case of `DSR` (Fig. 15(b)) is special, as its performance and latency are primarily constrained by disk performance. The elastic programs we used have very low disk activity, mainly for loading



(a) EMC(120) (b) EMC(150) (c) EMC(200) (d) EMC+(20) (e) EMC+(35) (f) EMC+(50)

FIGURE 14 | Normalized per-application speedup and slowdown for the different variants of the elasticity manager



(a) data-caching (b) data-serving-relational (c) nginx (d) NodeJS

FIGURE 15 | Percentage of client-side monitoring intervals with QoS violations for workloads including a latency-sensitive program.

input data and storing final computation results. However, this still impacts DSR, causing QoS violations even under Stock-Linux, which does not explicitly address QoS enforcement under I/O contention, just like EMC and EMC+. Nevertheless, EMC+(20)'s QoS-violation figures match Stock-Linux counterparts here.

We now turn our attention to throughput improvements of individual workloads w.r.t. Stock-Linux, as shown in Fig. 16. The throughput fluctuations across workloads primarily depend on (1) the amount of idle core cycles left by the inelastic application, (2) the elastic program's ability to increase performance by making opportunistic use of idle cores, and (3) the performance degradation caused to the inelastic program – reported in Fig. 14. In general, EMC(120) is the scheme providing the highest throughput improvements w.r.t. Stock-Linux (23% on average, as shown in Fig. 17(c)). This results from its aggressiveness in populating idle cores, which substantially accelerates elastic programs. EMC+(20) and EMC+(35) are the best-performing strategies in terms of QoS and the performance degradation of the inelastic program. Notably, these two approaches still yield substantial gains vs. Stock-Linux in average speedup (43%) and throughput (21%) –just slightly inferior figures to those delivered by EMC. Moreover, in most cases EMC+(20) incurs QoS violations in 5% or less of the monitoring intervals. This means that the top 1% of the total requests – see target RPS in Table 3 – during those few intervals (or top 5% in the case of `data-serving-relational`) exceed the maximum latency. As a result, the overall percentage of QoS violations over the total requests issued during each full experiment is much smaller; in practice, it is no greater than 1% for most workloads. Hence, EMC+(20) is the best choice in terms of QoS enforcement.

Second evaluation scenario (workloads W1–W25). Having identified the best-performing EMC and EMC+ configurations, we now proceed to discuss the results with the 4-application workloads. For simplicity, Figures 18 to 20 show the results just for Stock-Linux, EMC(120) – the configuration that yields the highest speedup of the elastic program, by aggressively utilizing idle

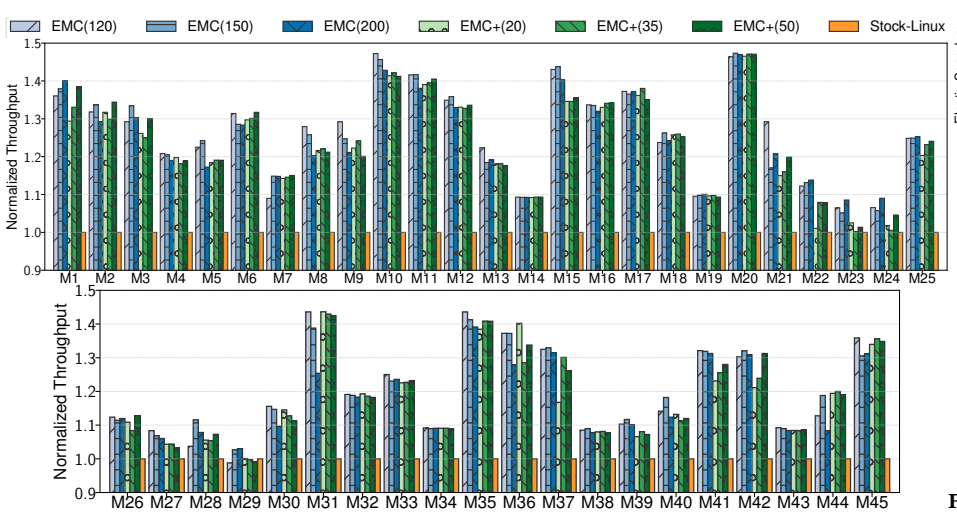


FIGURE 16 | Normalized throughput achieved by the different elasticity strategies

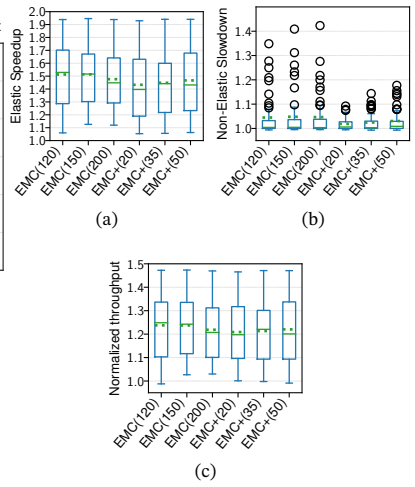


FIGURE 17 | Speedup, slowdown and throughput distribution of the various strategies. Mean values are represented with dotted green lines.

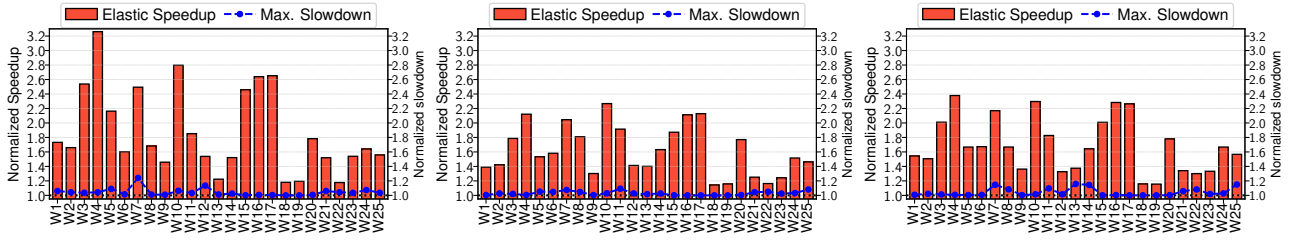


FIGURE 18 | Normalized per-application speedup and slowdown in the W1-W25 workloads under the different elasticity strategies

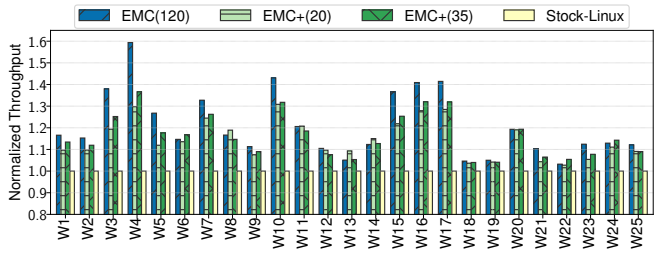


FIGURE 21 | Normalized throughput achieved for W1-W25

core cycles – and EMC+(20) and EMC+(35) – the EMC+ configurations that deliver the best trade-off between speedup and performance/QoS degradation.

Fig. 18 shows the normalized per-application speedup for the elastic application in the W1-W25 workloads. This figure also plots the maximum slowdown w.r.t. Stock-Linux observed across inelastic applications in the workload. As compared to the 2-application scenario, the elasticity manager now achieves much higher speedup figures, especially under EMC(120). Specifically, as the results in Figs. 17(a) and 20(a) reveal, EMC(120)'s average speedup rises from 1.51 (2-application workloads) to 1.88 (with 4 applications). This occurs because the elastic application may now potentially use a higher idle-core share of the system, "stealing" cores from up to three 16-core inelastic containers.

As for QoS and throughput results (Figs. 19 and 21, respectively), we observe trends similar to those in the 2-application scenario. EMC(120) delivers the best throughput figures for most workloads, whereas EMC+ sacrifices throughput (and especially opportunities for elastic application acceleration) in exchange for improved QoS – with EMC+(20) being the safest choice in that regard. The most critical observation is the fact that the average throughput gains are smaller as compared to the 2-application

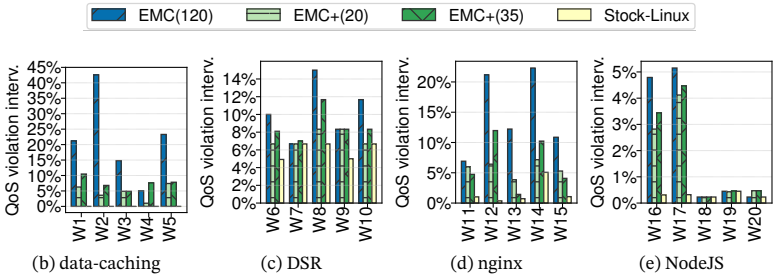


FIGURE 19 | Percentage of client-side monitoring intervals with QoS violations for workloads including a latency-sensitive program.

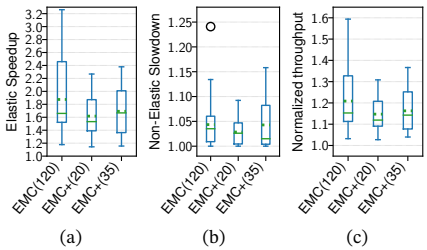


FIGURE 20 | Speedup, slowdown and throughput distribution across W1 to W25 workloads.

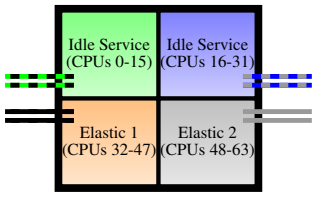


FIGURE 22 | Initial application mapping in experiments with 2 elastic applications

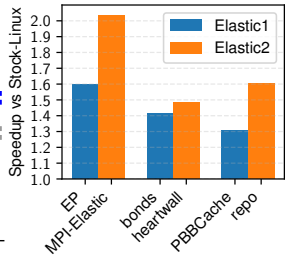


FIGURE 23 | Speedup of different elastic applications

scenario (see Figs. 17(c) and 20(c)). This outcome is a direct consequence of the inherent nature of the throughput metric itself (Eq. 1) whose normalized value tends to one as the proportion of applications experiencing a slowdown close to one increases[62]. In this specific case, 75% of the applications in the workload (i.e., the three inelastic programs) exhibit a slowdown near one, whereas in the 2-application scenario, 50% of the programs (at best) exhibit this slowdown value under EMC and EMC+.

7.5 | EMC+'s main limitations and future work

Our extensive evaluation – comprising 70 workloads and a wide range of applications– has demonstrated that the most conservative configuration of EMC+ is able to successfully accelerate different types of elastic applications in an opportunistic fashion. Crucially, it does so with minimal impact on the performance and QoS of inelastic applications. However, our proposal still has room for improvement. The limitations discussed next shed light on some of the potential enhancements planned for future work.

7.5.1 | Limitation #1. Core redistribution irrespective of performance benefit

When EMC+ detects idle core cycles in the system, it estimates how many cores to "steal" from inelastic applications, and then assigns all these cores to elastic applications in the workload. This course of action relies on the expectation that those cores will be effectively utilized by elastic applications. Notably, EMC+ (i) does not check whether a particular elastic application actually benefits from all the extra cores allotted, (ii) does not establish any cap on the number of extra cores that it can use and, (iii) in the presence of multiple elastic applications, it evenly distributes unused cores among all of them, which is not always the best choice from the system optimization standpoint.

To illustrate the differences in the performance benefits obtained by elastic applications, we ran three additional workloads, with two elastic applications each. Fig. 22 shows the initial per-application assignment in these additional four-application workloads. In this scenario, the first two quadrants of the multi-core CPU are used to run cloud-like services that remain idle during the experiment. Particularly, we launched `nginx` and `NodeJS` in these two sets of cores, but issued no client requests during the experiment. The goal of this scenario is to assess the acceleration potential of elastic applications when leveraging extra cores from idle containers, which EMC+ equally distributes among them.

Fig. 23 shows the relative speedup of each elastic application, where each pair of bars corresponds to a separate workload. The results reveal that each elastic application obtains a different speedup from using the extra cores allotted, and also show that the disparities are more pronounced in the first and last workloads. In light of these results, the main takeaways are as follows. First, by evenly distributing unused cores among applications, EMC+ misses opportunities to optimize system throughput. Favoring applications with better scalability in assigning extra cores would increase system throughput. Second, some applications –such as `bonds` or `PBBCache`– derive a modest benefit from running with a much larger set of cores. Therefore, setting an upper limit on the number of cores granted to an elastic application (based on its scaling potential) seems a reasonable decision to avoid wasting CPU resources unnecessarily.

Addressing these issues is challenging in the black-box scenario of this work, where applications do not provide the elasticity manager with performance-related feedback at runtime. However, leveraging application-specific performance models for runtime scalability estimation[36,21], together with online feedback from these models to EMC+, constitutes an interesting avenue for future work. Such an extension would enable us to further improve EMC+'s efficiency. Notably, the data structure shared between the application and the elasticity manager could be trivially augmented with an additional field, allowing the application to easily report the maximum number of cores it can effectively exploit according to its latest scalability predictions. This information could be used by EMC+ to perform a more efficient core redistribution, and also be taken into account by the cloud provider for billing purposes, since it could have a significant impact on the amount of CPU resources consumed under opportunistic elasticity.

7.5.2 | Limitation #2. Unawareness of shared-resource contention effects

Putting elasticity-exploitation aside, shared-resource contention effects are known to be one of the most relevant causes of performance degradation when running multiple applications on a multicore platform[63,1,4,3]. Co-running applications naturally compete for memory-related resources (space in shared caches or memory bandwidth) and for other system resources (e.g., disk and network bandwidth). This competition may lead to uneven and hard-to-predict performance degradation across applications, causing severe degradation of system-wide fairness and throughput, and potentially jeopardizing QoS enforcement. As stated in Sec. 2, none of the previous QoS-aware strategies designed to mitigate shared-resource contention effects exploit opportunistic elasticity, and, more importantly, unlike our black-box approach, most of them require receiving continuous feedback on application-level metrics from QoS-constrained applications to function[4,3,11]. This observation, together with the fact that our proposal is built as an OS-level elasticity layer on top of the contention-agnostic Linux scheduler[1,63], suggests that creating a contention-conscious version of EMC+ becomes a prioritary path for future research.

To shed light on the impact of opportunistic elasticity on shared-resource contention, we analyze a relevant workload scenario in our experiments: the M32 workload, which combines two memory-bandwidth intensive applications – `DeepSpeech` (inelastic) and `sradv2` (elastic). As shown in Fig. 14(d), the inelastic application suffers the highest slowdown (9.02%) under EMC+(20) – the most conservative configuration of our proposal. Figs. 24 and 25 show the memory bandwidth consumption of

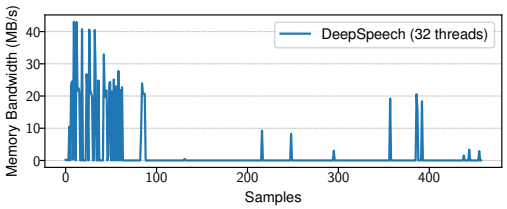


FIGURE 24 | Memory bandwidth consumption of `DeepSpeech`

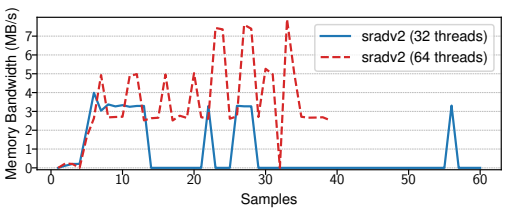


FIGURE 25 | Memory bandwidth consumption of `sradv2`

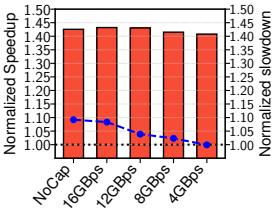


FIGURE 26 | Effects of bandwidth-consumption limiting in workload M32 under EMC+(20)

each application gathered every 120ms as they run alone on the system. Particularly, Fig. 25 reveals that the bandwidth consumption of the elastic application grows substantially with the number of threads. The main problem is that, in opportunistically increasing the number of workers of `sradv2` under EMC+, the associated increase in its memory-bandwidth consumption sometimes exhausts the bandwidth available through the memory channels shared with the inelastic application. This source of bandwidth contention is the cause of the performance degradation observed in the inelastic application.

A potential way to address this issue would be to limit the maximum number of workers in `sradv2` when high bandwidth consumption is detected. Unfortunately, `sradv2` exhibits brief high bandwidth spikes that go unnoticed unless very fine-grained bandwidth monitoring is used. As demonstrated in previous work [64], detecting these sharp fluctuations in memory-bandwidth consumption (crucial to detect severe contention scenarios) typically requires sampling intervals on the order of milliseconds. Because this monitoring granularity introduces excessive overhead in OS-level solutions, we explored an alternative mitigation method: leveraging the hardware support present in our platform for memory-bandwidth limitation [65]. This support, which has to be configured on a per-CCX basis, allows us to set an upper cap on per-application bandwidth consumption.

Fig. 26 shows the normalized performance boost and degradation (for the elastic and inelastic program, respectively) achieved by EMC+(20) when enforcing a different limit on memory bandwidth consumption, as compared to the default setting, where no limit is applied (labeled "NoCap"); note that the bandwidth limit only applies to the memory traffic through the channels associated with the cores where the inelastic application runs, as depicted in Fig. 13(a). The results reveal that limiting bandwidth consumption allows us to completely remove `DeepSpeech`'s performance degradation w.r.t. Stock-Linux (for the 4GBps cap) without significantly impacting the performance gains of the elastic application. This observation suggests that the dynamic exploitation of this hardware feature is a promising method to mitigate the impact of shared-resource contention in a future contention-conscious version of EMC+.

As pointed out in Sec. 7.4, I/O bandwidth contention becomes apparent in workloads including the disk-intensive `data-serving-relational` (DSR) application, based on PostgreSQL (see results in Fig. 15(b)). However, the negative impact of disk-related contention on QoS is similar under Stock-Linux and EMC+(20), which underscores that the problem arises even under conservative elasticity decisions. As for future work, we will explore the exploitation of OS-provided mechanisms for disk-bandwidth throttling (such as those used in [3,4]) as a means to address this source of contention under opportunistic elasticity.

7.5.3 | Limitation #3. Current applicability to virtual environments

Our EMC+ implementation supports workloads running natively on top of the OS or within a container. However, delivering opportunistic elasticity to applications running inside virtual machines (VMs) is not yet supported. Essentially, the version of the PMCSched framework [56], which EMC+'s implementation relies on, does not yet allow dealing with VMs within resource management plugins. The necessary VM-related support in PMCSched is currently under development. This upcoming support (KVM-based) relies on the fact that each VM is exposed to the Linux scheduler as a multi-threaded process, where each virtual CPU (VCPU) of the VM is a thread of this process [1]. So, a future VM-enabled version of EMC+ would manage each VM in much the same way as a multithreaded process.

Dealing with inelastic applications within VMs would be almost straightforward in EMC+'s implementation with this upcoming support. However, managing VM-based elastic applications poses a few challenges due to the semantic gap between the host OS or virtual machine monitor (VMM), and the guest OS. In particular, the shared-memory-based communication scheme that EMC+ exploits to interact with inelastic applications would not work directly if the elastic application runs within a VM, as the guest OS acts as an isolation layer preventing this direct communication from taking place. This limitation can be trivially avoided by running just elastic workloads natively on the host OS. However, investigating how to extend a guest OS to allow seamless interaction between EMC+ and a VM-based elastic application constitutes a promising avenue for future work.

8 | Conclusions

In this article we propose EMC+, an OS-level elasticity manager for containerized applications that targets multi-application workloads combining cloud and CPU-intensive elastic applications. EMC+ improves system throughput in multicore server systems by opportunistically using idle core cycles left by regular applications (ranging from client-server workloads to long-running batch programs) to run threads of elastic applications. These elastic applications have the ability to dynamically adapt their number of active workers to the current amount of allotted CPU resources. Our extensive experimental evaluation shows that EMC+ substantially boosts system throughput by effectively accelerating elastic HPC workloads, while ensuring minimal impact on the performance and QoS of cloud applications.

In addition to the new scalable container support – crucial to allow the execution of a broad range of multithreaded and multi-process cloud and HPC applications –, a key innovation of our EMC+ proposal lies in its novel design features (see Sec. 6.1) aimed at reducing migration-related overheads and smoothing the aggressiveness in exploiting idle core cycles left by inelastic applications. As our experiments reveal, the EMC elasticity strategy – a container-enabled extension of our earlier throughput-optimized EM proposal[25] created for comparison purposes –, achieves higher throughput gains, sometimes at the expense of severe performance/QoS degradation of inelastic applications. Our proposed EMC+ strategy successfully addresses this issue, ensuring minimal performance/QoS impact for a wide range of cloud applications (unsupported by EM[25]).

As for the practical applicability of our proposed approach in cloud datacenters, we primarily consider two main use cases: (1) the cost-effective execution of HPC and scientific applications leveraging elasticity, and (2) the exploitation of idle periods to opportunistically run CPU-intensive cloud-provider workloads.

Our experimental evaluation in Sec. 7 demonstrated that EMC+ is suitable for the first use case, making it possible to accelerate elastic HPC workloads while minimally affecting the performance and QoS of the remaining applications on the system. Therefore, our proposal lays the foundations of a novel cost-effective and highly efficient environment for elastically running medium- and small-scale HPC and scientific workloads in the cloud, serving as a middle ground between current fixed-resource container/VM instances for opportunistic yet inelastic execution of batch applications[40,41] and expensive high-end solutions for HPC in the cloud[24] (e.g., private clouds or on-premises clusters). Because elastic containers leverage CPU cores opportunistically rather than in an exclusive fashion, our proposal opens the door to affordable and efficient execution of long-running CPU-intensive workloads (e.g., extensive simulations with malleability support), making it possible to cope with unexpected peaks of computational demand in research institutions. Notably, as we discussed in Sec. 7.5.1, establishing limits on the maximum number of cores an elastic application is allowed to use, may play an important role in both the system throughput achieved and the price of the associated elasticity services in the cloud.

For the second use case, we envision using EMC+ for the opportunistic execution of any parallel CPU-intensive workload, including applications without malleability support. It is well known that cloud-providers run low-priority best-effort (BE) tasks together with critical workloads, as a means to improve resource usage during periods of low server utilization[9,3,66]. These BE tasks include batch analytics, AI training and inference, deep-learning-based image classification, video transcoding, CI/CD flows, etc.[42,66,40,41]. Notably, unlike the elastic HPC workloads considered in this work, BE tasks do not typically support fine-grained vertical elasticity, and may be temporarily stalled, restarted or relocated. To allow the opportunistic execution of BE tasks on a shared server, EMC+ could be complemented with a resource manager that decides which BE tasks to run, stop, resume or cancel based on the number of available idle cores that EMC+ dynamically reports as available. Designing such a resource manager and evaluating its effects on system throughput and QoS enforcement represent a valuable direction for future research.

ACKNOWLEDGMENTS

This work has been supported by the Spanish MCIU under Grants PID2021-126576NB-I00 and PID2024-158311NB-I00, funded also by MCIU/AEI/10.13039/501100011033 and "ERDF A way of making Europe".

References

1. Hu X, et al. Cacheman: A Comprehensive Last-Level Cache Management System for Multi-tenant Clouds. In: *Proc. of PPoPP '26*; 2026: 329–341
2. Shahrad M, Elnikety S, Bianchini R. Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds. In: *Proc. of SoCC'21*; 2021: 319–334.
3. Chen S, Delimitrou C, Martínez JF. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In: *Proc. of ASPLOS '19*; 2019: 107–120.
4. Lo D, et al. Heracles: Improving Resource Efficiency at Scale. In: *Proc. of ISCA'15*; 2015: 450–462.
5. Silva dVS, et al. Smart resource allocation of concurrent execution of parallel applications. *Concurr. Comput. Pract. Exp.* 2023; 35(17): e6600. doi: 10.1002/cpe.6600
6. Bilbao C, Saez JC, Prieto-Matias M. Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores. *IEEE Trans. Parallel Distrib. Syst.* 2023; 34(11): 2928–2945. doi: 10.1109/TPDS.2023.3309999
7. AMD . AMD EPYC 9965. https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html; . Accessed: 2026-07-13.
8. Al-Dhuraibi Y, et al. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Trans. Serv. Comput.* 2018; 11(2): 430–447. doi: 10.1109/TSC.2017.2711009
9. Zhao L, et al. Component-distinguishable Co-location and Resource Reclamation for High-throughput Computing. *ACM Trans. Comput. Syst.* 2024; 42(1–2): 2. doi: 10.1145/3630006
10. Zhong Y, et al. Managing Memory Tiers with CXL in Virtualized Environments. In: *Proc. of OSDI'24*; 2024: 37–56.
11. Patel T, Tiwari D. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In: *Proc. of HPCA '20*; 2020: 193-206.
12. Huang H, et al. Towards Exploiting CPU Elasticity via Efficient Thread Oversubscription. In: Proc. of HPDC'21. ; 2021: 215–226.
13. Herbst NR, Kounev S, Reussner R. Elasticity in cloud computing: What it is, and what it is not. In: *Proc. of ICAC '13*; 2013: 23–27.
14. Amazon . Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/; . Accessed: 2026-07-13.
15. DigitalOcean . DigitalOcean Elasticity. https://www.digitalocean.com/community/tools/digitalocean-elasticity; . Accessed: 2026-07-13.
16. Jeong B, Jeong YS. Autoscaling techniques in cloud-native computing: A comprehensive survey. *Computer Science Review* 2025; 58: 100791. doi: 10.1016/j.cosrev.2025.100791
17. Galante G, Righi R, Andrade C. Extending parallel programming patterns with adaptability features. *Cluster Computing* 2024; 27: 12547-12568. doi: 10.1007/s10586-024-04622-0
18. Kubernetes . Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/; 2026.
19. Anyscale . KubeRay Autoscaling. https://docs.ray.io/en/latest/cluster/vms/user-guides/configuring-autoscaling.html; 2026.
20. Finol G, et al. Exploiting inherent elasticity of serverless in algorithms with unbalanced and irregular workloads. *J. Parallel Distrib. Comput.* 2024; 190(C). doi: 10.1016/j.jpdc.2024.104891
21. Tarraf A, et al. Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities. *IEEE Trans. Par. Distrib. Syst.* 2024; 35: 1-14. doi: 10.1109/TPDS.2024.3406764

22. Galante G, Rosa Righi dR. Adaptive parallel applications: from shared memory architectures to fog computing (2002–2022). *Cluster Computing* 2022; 25(6): 4439-4461. doi: 10.1007/s10586-022-03692-2
23. Iordache A, et al. Resilin: Elastic MapReduce over Multiple Clouds. In: *Proc of CCGrid '13*; 2013: 261-268.
24. Netto MAS, et al. HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Comput. Surv.* 2019; 51(1). doi: 10.1145/3150224
25. Rubio J, et al. Exploiting Elasticity via OS-Runtime Cooperation to Improve CPU Utilization in Multicore Systems. In: *Proc. of PDP '24*; 2024: 35-43.
26. Cho Y, Guzman CAC, Egger B. Maximizing System Utilization via Parallelism Management for Co-Located Parallel Applications. In: *Proc. of PACT '18*; 2018: 1:1–1:14.
27. Iserte S, et al. Resource optimization with MPI process malleability for dynamic workloads in HPC clusters. *Future Generation Computer Systems* 2026; 174: 107949. doi: 10.1016/j.future.2025.107949
28. Martín-Álvarez I, et al. Dynamic spawning of MPI processes applied to malleability. *Int. J. High Perform. Comput. Appl.* 2024; 38(2): 69–93. doi: 10.1177/10943420231176527
29. Ahmad H, et al. Towards Resource-Efficient Reactive and Proactive Auto-Scaling for Microservice Architectures. *Journal of Systems and Software* 2025; 225: 112390. doi: 10.1016/j.jss.2025.112390
30. Chouliaras S, Sotiriadis S. Auto-Scaling Containerized Cloud Applications: A Workload-Driven Approach. *Simulation Modelling Practice and Theory* 2022; 121: 102654. doi: 10.1016/j.simpat.2022.102654
31. Ju L, Singh P, Toor S. Proactive Autoscaling for Edge Computing Systems with Kubernetes. In: *Proc. of UCC '21*; 2021: 1–8
32. Chanikaphon T, Amini Salehi M. UMS: Live Migration of Containerized Services across Autonomous Computing Systems. In: *Proc. of GLOBECOM '23*; 2023: 467–472
33. Liu M, et al. Low-Latency Layer-Aware Proactive and Passive Container Migration in Meta Computing. In: *Proc. of ICMC' 24*; 2024: 176–185
34. Iserte S, et al. DMRlib: Easy-Coding and Efficient Resource Management for Job Malleability. *IEEE Trans. Comp.* 2021; 70(9): 1443-1457. doi: 10.1109/TC.2020.3022933
35. Marques SMV, et al. Optimizing the EDP of OpenMP applications via concurrency throttling and frequency boosting. *J. Syst. Archit.* 2022; 123: 102379. doi: 10.1016/j.sysarc.2021.102379
36. Sanchez-Checa P, et al. Combining Malleability and Distributed Control Mechanisms to Reduce I/O Contention. In: *High Performance Computing*; 2026: 387–400
37. Pickartz S. *Virtualization as an Enabler for Dynamic Resource Allocation in HPC*. E.ON Energy Research Center, RWTH Aachen University . 2019.
38. AlibabaCloud . Building a High Performance Container Solution with Super Computing Cluster and Singularity. https://www.alibabacloud.com/blog/594569; 2019.
39. Agulló F, et al. Enhancing the output of time series forecasting algorithms for cloud resource provisioning. *Future Gen. Comput. Syst.* 2025. doi: 10.1016/j.future.2025.107833
40. Amazon . Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html; 2026. Accessed: 2026-07-03.
41. Microsoft . Azure Spot Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms; 2026. Accessed: 2026-07-03.
42. Caprara A, et al. Data center workload flexibility for power system demand response: Evidence from Alibaba traces. *International Journal of Electrical Power & Energy Systems* 2026; 178: 111940. doi: 10.1016/j.ijepes.2026.111940
43. Villarrubia J, et al. Solving the task scheduling and GPU reconfiguration problem on MIG devices via deep reinforcement learning. *Future Generation Computer Systems* 2026; 176: 108145. doi: 10.1016/j.future.2025.108145
44. PARSA-EPFL . Data Caching Benchmark. https://github.com/parsa-epfl/cloudsuite/blob/main/docs/benchmarks/data-caching.md; . Accessed: 2026-07-13.
45. PARSA-EPFL . Cloudsuite 4.0: A Benchmark Suite for Cloud Services. https://github.com/parsa-epfl/cloudsuite; . Accessed: 2026-07-13.
46. Roca A, et al. A Linux Kernel Scheduler Extension for Multi-core Systems. In: *Proc. of HiPC '19*; 2019: 353-362.
47. Bilbao C, Saez JC, Prieto-Matias M. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake. *Concurr. Comput. Pract. Exp.* 2023; 35(25): e7814. doi: 10.1002/cpe.7814
48. Velten M, et al. Memory Performance of AMD EPYC Rome and Intel Cascade Lake SP Server Processors. In: *Proc. of ICPE '22*; 2022: 165–175
49. AMD . HPC Tuning Guide for AMD EPYC 7002 Series Processors. https://docs.amd.com/v/u/en-US/amd-epyc-7002-tg-hpc-56827; 2020.
50. OpenBenchmarking . NGINX Benchmark. https://openbenchmarking.org/test/pts/nginx; . Accessed: 2026-07-13.
51. OpenBenchmarking . Node.js Express Load Test. https://openbenchmarking.org/test/pts/nginx; . Accessed: 2026-07-13.
52. OpenBenchmarking . DeepSpeech benchmark. https://openbenchmarking.org/test/pts/deepspeech; . Accessed: 2026-07-13.
53. Grauer-Gray S, et al. Accelerating financial applications on the GPU. In: *Proc. of GPGPU '13*; 2013: 127–136.
54. Garcia-Garcia A, et al. PBBCache: an open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *Journal of Computational Science* 2020: 101102. doi: 10.1016/j.jocs.2020.101102
55. IPyparallel. Using IPython for parallel computing. https://ipyparallel.readthedocs.io/; . Accessed: 2026-07-14.
56. Bilbao C, Saez JC. PMCTrack v4.0 release. https://github.com/jcsaezal/pmctrack/releases/tag/v4.0; . Accessed: 2026-07-13.
57. Gregg B. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional . 2019.
58. Bilbao C, Saez JC. PMCSched website. https://pmctrack-linux.github.io/pmcsched; . Accessed: 2026-07-15.
59. Eyerman S, Eeckhout L. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters* 2014; 13(2): 93-96. doi: 10.1109/L-CA.2013.9
60. Lu X, et al. CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning. In: *Proc. of HPCA'24*; 2024.
61. Shelepov D, et al. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.* 2009; 43(2): 66–75. doi: 10.1145/1531793.1531804
62. Saez JC, et al. LFOC+: A Fair OS-Level Cache-Clustering Policy for Commodity Multicore Systems. *IEEE Trans. Comp.* 2022; 71(8): 1952-1967. doi: 10.1109/TC.2021.3112970
63. Aznal J, Saez JC, Bilbao C. RDPart: A reuse-based OS-level cache-partitioning policy for fairness optimization in cloud data centers. In: *Proc. of ICPP '26*; 2026: To appear. https://pmctrack-linux.github.io/assets/papers/rdpart-icpp26.pdf.
64. Xu D, Wu C, Yew PC. On Mitigating Memory Bandwidth Contention Through Bandwidth-aware Scheduling. In: *Proc. of PACT '10)*; 2010: 237–248.
65. AMD . AMD64 Technology Platform QoS Extensions. https://developer.amd.com/wp-content/resources/56375.pdf; .
66. Tao Y, Liu J. Resource allocation and task scheduling policy for co-located clusters. *Journal of Parallel and Distributed Computing* 2026; 215: 105307. doi: 10.1016/j.jpdc.2026.105307